HELHa
Haute École Louvain en Hainaut

Syllabus

Développement de jeux vidéo C++

Bachelier informatique de gestion Mons
BLOC 3 - UE 308

V. Altares

Année académique 2023 – 2024

ÉCONOMIQUE

1.Éléments préalables

Le langage C++, surnommé au départ « C with classes », a été conçu à partir de 1979 par Bjarne Stroustrup (AT&Bell Laboratories) avec l'objectif d'ajouter les possibilités de la programmation objet (POO) au langage C.

Depuis sa fondation C++ s'est maintenu parmi les trois langages les plus populaires (Java, C, C++).

La première normalisation du langage C++ remonte à 1998. La normalisation de 1998 standardise la base du langage (*Core Language*) ainsi que la « *C++ Standard Library* ». Le standard actuel a été ratifié en 2011. Depuis, des mises à jour sont publiées régulièrement : en 2014 (ISO/CEI 14882:2014, ou C++14), en 2017 (ISO/CEI 14882:2017, ou C++17) puis en 2020 (ISO/IEC 14882:2020, ou C++20).

La bibliothèque standard du C++ est une bibliothèque de classes et de fonctions standardisées pour le langage C++, elle contient également le bibliothèque standard du C ainsi que la STL (standard template library).

Les principaux avantages du C++ sont les suivants :

- performances du C ;
- grand nombre de fonctionnalités ;
- facilité d'utilisation des langages objets ;
- portabilité des fichiers sources ;
- facilité de conversion des programmes C en C++, et, en particulier, possibilité d'utiliser toutes les fonctionnalités du langage C ;
- contrôle d'erreurs accru.

La richesse du contrôle d'erreurs du langage, basé sur un typage très fort, permet de signaler un grand nombre d'erreurs à la compilation.

Le C++ peut donc être considéré comme un « super C ».

Néanmoins, il existe certaines incompatibilités entre le C et le C++.

Les programmes C ne se compilent pas toujours directement en C++. Quelques adaptations sont donc souvent nécessaires, cependant, elles restent minimes, puisque la syntaxe du C++ est basée sur celle du C.

La norme C99 a pour but de diminuer les incompatibilités entre le C et le C++.

Tous les programmes C peuvent être corrigés pour compiler à la fois en C et en C++.

2.Spécificités du C++

2.1.Les commentaires en C++

Il existe deux types de commentaires en C++ :

- les commentaires de type C
- les commentaires de fin de ligne (qui ne sont disponibles qu'en C++).

En C (C90) :

```
/*  Ceci est un commentaire C, qui peut s'étendre sur
    plusieurs lignes   */
```

En C++ :

```
//  Ceci est un commentaire C++ qui s'arrête à la fin de la ligne
```

2.2.Entrées-sorties simples

C++ dispose de nouvelles possibilités d'entrées-sorties. Ces fonctionnalités étant détaillées plus loin, nous n'envisageons ici que l'utilisation simple des flux standards entrées-sorties, c'est-à-dire la manière de faire en C++ les opérations que l'on fait habituellement en C avec les fonctions printf et scanf.

Un programme qui utilise les flux standard d'entrée-sortie doit comporter une directive :

```
#include <iostream>
```

Et au moins une des déclarations suivantes:

```
using std::cout;
using std::cin;
using std::cerr;
```

La directive

```
using namespace std ;
```

peut remplacer les déclarations précédentes.

Les flux d'entrée-sortie sont représentés dans les programmes par les trois objets, pré-déclarés et pré-initialisés, suivants :

- cin, le flux standard d'entrée (l'équivalent du stdin de C), qui est habituellement associé au clavier,
- cout, le flux standard de sortie (l'équivalent du stdout de C), qui est habituellement associé à l'écran,
- cerr, le flux standard pour la sortie des messages d'erreur (l'équivalent du stderr de C), également associé à l'écran du poste de travail.

Les lectures et écritures sur ces unités ne se font pas en appelant des fonctions, mais

à l'aide des deux opérateurs

- <<, appelé *opérateur d'injection* (injection de données dans un flux de sortie),
- >>, appelé *opérateur d'extraction* (extraction de données d'un flux d'entrée).

La grande commodité d'utilisation de ces opérateurs repose sur la surcharge des opérateurs (voir plus loin) qui permet la détection des types des données à lire ou à écrire, ce qui autorise, le programmeur à ne pas s'encombrer avec des spécifications de format.

La syntaxe d'une injection de données sur la sortie standard cout est :

cout << *expression à écrire ;*

le résultat de cette expression est l'objet cout lui-même. On peut donc lui injecter une autre donnée, puis encore une, etc :

((cout << *expression*) << *expression*) << *expression ;*

Puisque l'opérateur << est associatif à gauche, on écrira beaucoup plus simplement :

cout << *expression* << *expression* << *expression ;*

Le même procédé existe avec l'extraction sur l'objet cin.

Par exemple, le programme suivant est un programme C++ complet qui permet de permuter et d'afficher à l'écran deux nombres saisis au clavier.

```
#include <iostream>
using namespace std;
void permuterCpp(int *a, int *b)
{
	int save = *a;
	*a = *b;
	*b = save;
}
int main()
{
	int i, j ;
	cout<<"Encodez le premier nombre : ";
	cin>>i;
	cout<<"Encodez le second nombre: ";
	cin>>j;
	permuterCpp(&i,&j);
	cout<<"En C++ "<<i<<", "<<j<<endl;
	return 0;
}
```

2.3.Placement des déclarations de variables

En C (C90) les déclarations de variables doivent être placées au début d'un bloc. En C++, cela n'est plus obligatoire et on peut mettre une déclaration de variable partout où

on peut mettre une instruction.

Cette nouvelle possibilité permet d'éviter d'employer des variables déjà déclarées mais non encore initialisées.

Exemple :

Version C :

```
int j;

...
for (j = 0; j < 10; j++)
…;
```

Version C++ :

```
for (int j = 0; j < 10; j++)
…;
```

La norme C99 a ouvert la possibilité de déclarer des variables partout dans le code comme en C++.

2.4.Références

2.4.1.Notion de référence

A côté des pointeurs, les références sont une autre façon de manipuler les adresses des objets placés dans la mémoire. Les références sont des synonymes (alias) d'identificateurs. Une référence est un pointeur géré de manière interne par la machine.

Si T est un type donné, le type « référence sur T » se note T&.

Une référence doit toujours se référer à un identificateur : on ne peut donc déclarer une référence sans l'initialiser. La déclaration d'une référence ne crée pas un nouvel objet . Les références se rapportent nécessairement à des identificateurs déjà existants.

La syntaxe de la déclaration d'une référence est la suivante :

```
type &ref = identificateur;
```

Après cette déclaration, ref peut être utilisé partout où identificateur peut l'être. Ce sont des synonymes.

Exemple :

```
int i = 0;
int &ref = i;                    // Référence sur la variable i.
ref = ref + i;                   // Double la valeur de i (et de ref).
```

Comme on le voit dans l'exemple ci-dessus, une référence est une adresse mais, hormis lors de son initialisation, *toute opération effectuée sur la référence agit sur l'objet référencé, non sur l'adresse.*

Il est possible de faire des références sur des valeurs numériques. Les références doivent alors être déclarées comme constantes, puisqu'une valeur numérique est une constante :

```
const int &ref = 7;                    // Référence sur 7.
int &mauvais = 7;                      // Quid?
```

L'opérateur & (à un argument) a une signification très différente selon le contexte dans lequel il apparaît :

- employé dans une déclaration, comme dans int &r = x; il sert à indiquer un type référence : « r est une référence sur un int »
- employé ailleurs que dans une déclaration il indique l'opération « obtention de l'adresse », comme dans l'expression p = &x qui signifie « affecter l'adresse de x à p;
- en C++, comme en C, l'opérateur & binaire (à deux arguments) exprime la conjonction bit-à-bit de deux mots-machine.

2.4.2.Passage de paramètres par référence

En C, on ne peut passer des variables en paramètre dans une fonction que d'une seule manière, c'est le *passage par valeur (*les pointeurs permettent de simuler un passage par adresse en passant une adresse par valeur !).

Dans le passage par valeur, la valeur de l'expression (paramètres effectifs) passée en paramètre est copiée dans une variable locale (paramètres formels). C'est cette variable qui est utilisée pour faire les calculs dans la fonction appelée.

Si l'expression passée en paramètre est une variable, son contenu est copié dans la variable locale. Aucune modification de la variable locale dans la fonction appelée ne modifie la variable passée en paramètre, parce que ces modifications ne s'appliquent qu'à une copie de cette dernière.

C++ permet de passer les variables elles-mêmes en utilisant des paramètres sous la forme de références.

Dans ce cas, toute modification du paramètre dans la fonction appelée entraîne la modification de la variable passée en paramètre.

L'intérêt essentiel des références est donc de permettre de passer aux fonctions des paramètres modifiables qui garderont leur nouvelles valeurs après l'appel de la fonction, sans utiliser les pointeurs.

Exemple :

En C :

```
void permuter(int *a, int *b)
{
    int save = *a;
```

```
    *a = *b;
    *b = save;
}
```

L'utilisation de pointeurs force à invoquer permuter(&u,&v).

En C++ :

```
void permuter(int &a, int &b)                    //choix du passage par référence
{
    int save = a;
    a = b;
    b = save;
}
```

Lorsque l'on invoque la fonction permuter(u,v), les paramètres formels a et b sont initialisés avec les adresses des paramètres effectifs u et v, mais cette utilisation des adresses reste cachée, le programmeur n'a pas à s'en occuper.

Autrement dit, à l'occasion d'un tel appel, a et b ne sont pas des variables locales de la fonction recevant des copies des valeurs des paramètres, mais d'authentiques alias des variables u et v. Il en résulte que l'appel ci-dessus permute effectivement les valeurs des variables u et v.

Cet exemple montre bien que le choix du mode de passage se fait dans la déclaration de la fonction et non dans son appel.

En Java les variables d'un type de base sont transmises par valeur, les objets sont transmis par référence.

2.4.3. Références sur des données constantes

Lorsque les arguments transmis sont des objets complexes dont la recopie exige beaucoup d'espace mémoire, le passage par référence s'avère plus efficace que le passage par valeur.

Néanmoins, le passage par référence contrairement au passage par valeur ne garantit pas la sécurité des arguments transmis qui peuvent être modifiés.

Lorsque l'on déclare les arguments comme des références sur des objets non constants, à l'aide du qualifieur const, on garantit la sécurité tout en bénéficiant du passage par référence :

Exemple :

```
void uneFonction(const unType & arg)
{
…
}
```

arg n'est pas une recopie de l'argument effectif (puisque référence) mais il ne peut pas

être modifié par la fonction (puisque const).

2.5.Arguments par défaut des fonctions

En C, l'appel d'une fonction doit contenir autant d'arguments que la fonction en attend effectivement.

C++ autorise l'emploi d'arguments par défaut. Les paramètres formels d'une fonction peuvent avoir des valeurs par défaut.

Exemple :

```
void fonction(int a, bool def = true, int b1 = 1, int b2 = 2)
```

Lors de l'appel d'une telle fonction, les paramètres effectifs correspondants peuvent alors être omis (ainsi que les virgules correspondantes), les paramètres formels seront initialisés avec les valeurs par défaut.

Exemples :

```
fonction(8) ;
fonction(1,false) ;
fonction(4,true,50) ;
fonction(3,false,5,6) ;
```

Dans le prototype d'une fonction, les paramètres par défaut sont toujours à la fin de la liste des arguments.

Les arguments par défaut n'existent pas en Java.

2.6.Surcharge de fonctions (overloading)

Il est interdit en C de définir plusieurs fonctions qui portent le même nom.

C++, permet de définir plusieurs fonctions du même nom pour autant que le compilateur puisse les différencier. Le compilateur peut différencier deux fonctions de même nom en regardant la liste et le type des paramètres qu'elle reçoit.

Il est donc possible d'écrire des fonctions de même nom (on les appelle alors des surcharges) si et seulement si toutes les fonctions portant ce nom peuvent être distinguées par leurs listes et leurs types d'arguments (et non par leur valeur de retour).

La fonction qui sera appelée sera celle dont la liste est la plus proche des valeurs passées en paramètre lors de l'appel.

Exemple :

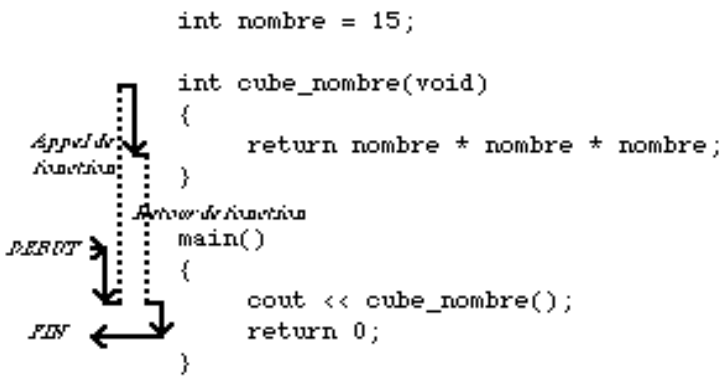
```
void fonction(int a,int b)
{
      cout<<"Deux entiers "<<a<<", "<<b<<endl;
}
```

```
void fonction(int a,float b)
{
    cout<<"Un entier et un float "<<a<<", "<<b<<endl;
}
```

S'il n'y a pas de correspondance exacte, alors des règles parfois complexes s'appliquent, pour déterminer la fonction à appeler. Malgré ces règles, il existe de nombreux cas de figure ambigus, que le compilateur ne peut pas résoudre.

2.7.Fonctions en ligne

Habituellement, un appel de fonction occasionne une *rupture de séquence* : à l'endroit où un appel est réalisé, la machine cesse d'exécuter séquentiellement les instructions en cours, et l'exécution continue ailleurs, là où se trouve le code de la fonction.



Dans une fonction en ligne, il n'y a pas, du moins en principe, de rupture de séquence lors de l'appel. Au lieu de cela, le compilateur remplace l'appel de la fonction par le corps de celle-ci, en mettant les arguments effectifs à la place des arguments formels.

Un tel traitement ne peut convenir qu'à des fonctions de petite taille fréquemment appelées (sinon le code compilé risque de devenir démesurément volumineux) et rapides (si une fonction effectue une opération lente, le gain de temps obtenu en supprimant l'appel est négligeable).

Les fonctions récursives ne peuvent être en ligne.

En C++ on indique qu'une fonction doit être traitée en ligne en faisant précéder sa définition par le mot inline :

Exemple :

```
inline int max(int i, int j)
{
    if (i>j) return i;
    return j;
}
```

Pour ce type de fonction, il est tout à fait justifié d'utiliser le mot clé inline.

Notons enfin, que puisque les fonctions inline sont insérées telles quelles aux endroits où elles sont appelées, il est nécessaire qu'elles soient complètement définies avant

leur appel. Par conséquent, de telles fonctions sont généralement écrites dans des fichiers en-tête qui doivent être inclus dans tous les fichiers comportant des appels de ces fonctions.

En fin de compte c'est le compilateur qui décide si la fonction sera en ligne ou non.

2.8.Opérateurs new et delete

Les fonctions malloc et free de la bibliothèque standard C sont disponibles en C++. Mais il est fortement recommandé d'utiliser les opérateurs new et delete.

Les objets créés à l'aide de new sont initialisés à l'aide des constructeurs correspondants, ce que ne fait pas malloc. De même, les objets libérés en utilisant delete sont détruits en utilisant le destructeur de la classe correspondante, contrairement à ce que fait free.

Pour allouer un unique objet :

new *type*

Pour allouer un tableau de n objets :

new *type* [n]

En cas de succès new renvoie l'adresse de l'objet crée, en cas d'échec il renvoie la valeur 0.

Si type est une classe possédant un constructeur par défaut, celui-ci sera appelé une fois (cas de l'allocation d'un objet simple) ou n fois (allocation d'un tableau d'objets) pour construire l'objet ou les objets alloués.

Exemples (on suppose que Personne est un type défini avec un constructeur par défaut) :

```
Personne *ptr = new Personne;          // un objet Personne
int *tab = new int [n] ;               // un tableau de n int
```

En C:

```
int *tab=(int*)malloc(n * sizeof (int));
```

Comme on peut s'y attendre, l'instruction

Personne *ptr = new Personne;

est illégale si la classe Personne ne possède pas de constructeur sans argument.

Exemples:

```
Point* pt=new Point(1,2);              // correct
Point* pp=new Point[11]();             // correct
Point* pp=new Point[11](1,2);          //  ERREUR
```

L'opérateur delete restitue la mémoire dynamique. Si la valeur de p a été obtenue par un appel de new, on écrit :

delete p;

dans le cas d'un objet qui n'est pas un tableau, et

delete [ ] p;

si ce que p pointe est un tableau.

En Java, l'opérateur new ne s'applique qu'aux objets.

2.9.Appel et définition de fonctions écrites en C

Pour que la compilation et l'édition de liens d'un programme avec des fonctions surchargées soient possibles, le compilateur C++ doit fabriquer pour chaque fonction un *nom long* comprenant le nom de la fonction et une représentation codée de sa signature; c'est ce nom long qui est communiqué à l'éditeur de liens.

Or, les fonctions produites par un compilateur C n'ont pas de tels noms longs ; si on ne prend pas de disposition particulière, il sera donc impossible d'appeler dans un programme C++ une fonction écrite en C, ou réciproquement. On remédie à cette impossibilité par l'utilisation de la déclaration « extern "C" » :

```
extern "C"   {
                 déclarations et définitions d'éléments
                 dont le nom est représenté à la manière de C
             }
```

2.10.Le type bool

Le type bool (pour booléen) comporte deux valeurs : false et true.

Contrairement à C (C90):

-le résultat d'une opération logique (&&, | |, etc.) est de type booléen,

-là où une condition est attendue on doit mettre une expression de type booléen et

-il est déconseillé de prendre les entiers pour des booléens et réciproquement.

Il sera donc préférable d'écrire :

if (a !=0) plutôt que

if (a).

La norme C99 comporte bel et bien un type bool comme en C++.

3.Classes

3.1.Classes et objets

Un objet est une entité cohérente regroupant des données et des fonctions.

Une classe est une sorte de moule à partir duquel on peut créer des objets.

Les classes sont les descriptions des objets (on dit que les classes sont la méta donnée des objets), lesquels sont des instances de leur classe.

L'implémentation d'un objet se fait donc en indiquant les données de l'objet et en définissant les fonctions qui peuvent lui être appliquées.

- dans la déclaration, le mot réservé class remplace le mot struct,
- les données ne sont normalement pas accessibles en dehors de la classe,
- certains champs de la classe sont des fonctions (méthodes).

On appelle

- objet (instance) une donnée d'un type classe ou structure,
- donnée membre (attribut) un membre qui est une variable,
- fonction membre (opération) un membre d'une classe qui est une fonction.

Le programme suivant est une première version d'une classe Point destinée à représenter les points affichés dans une fenêtre graphique :

```
class  Point
{
private:                                        // spécificateur d'accès
      int x;
      int y;
public:                                         // spécificateur d'accès
      void afficher()
      {
            cout << " (" << x << ", "  << y << " ) " ;
      }
      void placer(int abs, int ord)
      {
            x = abs;
            y = ord;
      }
};
```

L'association de données et de fonctions membres au sein d'une classe, avec la possibilité de rendre privés certains d'entre eux, s'appelle **l'encapsulation** des données.

Les spécificateurs d'accès private et public permettent de définir depuis quelle section du code on peut utiliser un membre.

Les coordonnées x et y d'un point sont privées, elles ne peuvent être modifiées autrement que par un appel de la fonction placer sur ce point.

De cette façon, on peut garantir aux utilisateurs de cette classe que tous les objets crées auront toujours des coordonnées correctes.

Chaque objet peut donc prendre soin de sa propre cohérence interne.

3.2.Accès aux membres

3.2.1.Accès aux membres d'un objet

On accède aux membres des objets en C++ comme on accède aux membres des structures en C.

Par exemple, à la suite de la définition de la classe Point donnée précédemment on peut déclarer des variables de cette classe en écrivant :

```
Point a, *p;        // un point et un pointeur de point
```

L'accès aux membres du point a s'écrit :

```
a. x = 0 ;                          // accès au membre x du point a
                                    // (non autorisé en dehors de la
                                    // classe puisque x est privé)
a.placer(3,4) ;                     // appel de la fonction placer
                                    // de l'objet a
```

Si le pointeur p a été initialisé, par exemple par une expression telle que

```
p = new Point;   // allocation dynamique d'un point
```

alors des accès analogues aux précédents s'écrivent :

```
p→x = 0 ; (équivalent à (*p).x=0)   // accès au membre x du point
                                    // pointé par p
p→placer(3,4) ;  // appel de la fonction placer
                                    // de l'objet pointé par pt
```

3.2.2.Accès à ses propres membres

Un objet accède à ses propres membres quand des membres de cet objet apparaissent dans une expression écrite dans une fonction du même objet.

Dans ce cas la notation se simplifie : on peut écrire le membre tout seul, sans expliciter l'objet en question.

Exemple :

```
class Point
{
```

```
…
void afficher()
     {
          cout << " (" << x << ", "  << y << " ) " ;
     }
…
} ;
```

Dans la fonction afficher, les membres x et y dont il est question sont ceux de l'objet à travers lequel on aura appelé cette fonction.

Autrement dit, lors d'un appel comme

unPoint.afficher(); le corps de cette fonction sera équivalent à

```
cout << " (" << unPoint.x << ", "  << unPoint.y << " ) " ;
```

3.2.3.Autoréférence : this

La variable this permet de faire référence à l'objet à travers lequel une méthode a été appelée.

Plus précisément, this représente un pointeur vers l'objet en question.

Exemple :

La fonction afficher pourrait se réécrire :

```
void afficher()
{
     cout << "(" << this→x << "," << this→y << ")";
}
```

3.2.4.Membres publics et privés

Par défaut, les membres des classes sont privés.

Les mots clés public et private permettent de modifier les droits d'accès des membres :

```
class nom
{
private:
     …   // les membres déclarés ici sont « private »
public  :
     …   // les membres déclarés ici sont « public »

                                                  // etc.
  };
```

Les expressions public: et private: peuvent apparaître un nombre quelconque de fois dans une classe.

L’usage veut que l’on commence par la déclaration des membres privés et que l’on termine par les membres publics.

Un membre public d'une classe peut être accédé partout où il est visible ; un membre privé ne peut être accédé que depuis une fonction membre de la classe (les notions de membre protégé, et de classes et fonctions amies, nuanceront cette affirmation).

Si p est un objet de type Point :

- dans une fonction qui n'est pas membre ou amie de la classe Point, les expressions p.x ou p.y constituent des accès illégaux aux membres privés x et y de la classe Point,
- les expressions p.afficher() ou p.placer(u, v) sont des accès légaux aux membres publics afficher et placer.

3.2.5.Encapsulation

Les fonctions membres d'une classe ont le droit d'accéder à tous les membres *de la classe :* deux objets de la même classe ne peuvent rien se cacher.

Par exemple, le programme suivant montre notre classe Point augmentée d'une fonction pour calculer la distance d'un point à un autre :

```
class Point
{
private:
    int x, y;
public:
    void afficher()
    {
        cout << " (" << x << ", "  << y << " ) " ;
     }
    double distance(Point autrePoint)
    {
        int dx = x - autrePoint.x;
        int dy = y - autrePoint.y;
        return sqrt(double(dx * dx + dy * dy));
    }
} ;
```

Lors d'un appel tel que p.distance(q) l'objet p accède aux membres privés x et y de l'objet q. On dit que C++ pratique l'encapsulation au niveau de la classe, non au niveau de l'objet.

3.2.6.Données membres statiques

Les données membres statiques n’existent qu’en un seul exemplaire, indépendamment des objets de la classe.

Envisageons la classe Point avec en plus une variable nombreDePoints qui renseigne le nombre de points crées.

```
class Point
{
      int x, y;
public :
      Point(int a, int b)
      {
            x = a;
            y = b;
            nombreDePoints++;
      }
      static int nombreDePoints;
} ;
```

Bien que chaque objet Point possède ses propres exemplaires des membres x et y, quel que soit le nombre de points existants à un moment donné, il n'existe qu'un seul exemplaire du membre nombreDePoints.

L'initialisation d'une variable membre statique se fait dans la *portée globale* (et donc en dehors de la classe), même s'il s'agit de membres privés :

```
int Point ::nombreDePoints = 0;
```

Cette instruction doit être écrite dans un fichier *.cpp , non dans un fichier *.h.

L'accès à un membre statique depuis une fonction membre de la même classe s'écrit comme l'accès à un membre ordinaire.

L'accès à un membre statique depuis une fonction non membre peut se faire à travers un objet, n'importe lequel, de la classe :

```
Point a, b, c;
cout << a.nombreDePoints <<endl;
```

L'accès peut s'écrire aussi plus lisiblement sans référence à un objet :

```
cout << Point::nombreDePoints <<endl;
```

A partir de C++ 17, l'initialisation de la variable statique peut se faire au moment de la déclaration de la manière suivante :

```
…
static inline nombreDePoints2017 = 0 ;
```

3.2.7.Fonctions membres statiques

Certaines fonctions membres d'une classe peuvent avoir un rôle totalement indépendant d'un quelconque objet.

Une fonction membre statique n'est pas attachée à un objet, son appel ne nécessite plus que le nom de la classe.

Par conséquent :

- elle ne dispose pas du pointeur this, de sa classe,
- elle ne peut référencer que les fonctions et les membres statiques.

Exemple :

```
class Point
{
    int x, y;
    static int nombreDePoints;
public:
    Point(int a, int b)
    {
        x = a;
        y = b;
        nombreDePoints++;
    }
    static int combienDePoints()
    {
        return nombreDePoints;
    }
} ;
```

Pour afficher le nombre de points existants on devra maintenant écrire une expression comme (p étant de type Point) :

```
cout << p.combienDePoints() << endl;
```

Expression à laquelle on préférera, une instruction qui ne fait pas intervenir de point particulier :

```
cout << Point::combienDePoints() << endl;
```

3.3.Affectation d'objets

L'affectation globale de deux objets est possible, tout comme elle l'était en C entre deux structures de même type.

```
Point a, b ;
a=b ;
```

provoquera la recopie des valeurs des membres x et y de b dans les membres correspondants de a. Ce type de copie est appelé « copie superficielle » car lorsque certaines données membres sont des pointeurs les zones pointées ne sont pas recopiées.

3.4.Constructeurs et destructeurs

3.4.1.Définition de constructeurs

Un constructeur d'une classe est une fonction membre particulière qui :

- a le même nom que la classe,
- n'indique pas de type de retour,

Le constructeur permet d'initialiser un objet, en donnant des valeurs à ses données membres.

Exemple :

```
class Point
{
…
public:
    Point(int abs, int ord)                    //constructeur Point
    {
        x = abs ;
        y = ord;
    }
 …
} ;
```

Un constructeur de la classe est toujours appelé, explicitement ou implicitement, lorsqu'un objet de cette classe est créé.

En C++, la création d'un objet se réalise par la déclaration et l'appel du constructeur. Le constructeur assure la cohérence des objets en évitant les variables non initialisées.

3.4.2.Surcharges d'un constructeur

Comme toute autre fonction membre, un constructeur peut être surchargé :

```
class Point
{
…
public:
    Point()
    {
        x=0 ;
        y=0 ;
    }
    Point(int abs)                                  //un constructeur
    {
        x=abs;
    }
    Point(int abs, int ord)                         //surcharge d'un
constructeur
    {
        x = abs;
        y = ord;
    }
…
```

```
} ;
```

L'emploi de paramètres avec des valeurs par défaut permet de grouper des constructeurs.

La classe suivante possède les mêmes constructeurs que la précédente :

```
class Point
{
…
public:
      Point(int abs = 0, int ord =0)
      {
            x = abs;
            y = ord;
      }
…
} ;
```

Comme les autres fonctions membres, les constructeurs peuvent être déclarés dans la classe et définis ailleurs (voir le point Exploitation des classes).

```
class Point
{
…
public:
      Point(int abs = 0, int ord = 0);
      …
} ;
```

et plus loin :

```
Point ::Point(int abs, int ord)
{
      x = abs;
      y = ord;
}
```

3.4.3.Appel des constructeurs

Un constructeur est toujours appelé lorsqu'un objet est crée, soit explicitement, soit implicitement. Les appels explicites peuvent être écrits sous deux formes :

```
Point a (1, 3);
Point b = Point(5, 7);
```

Un objet peut aussi être crée dynamiquement et être initialisé implicitement ou explicitement.

Cela s'écrit :

```
Point *p, *p2;
    p = new Point() ;
    p2 = new Point(2, 4) ;
```

Dès qu'un constructeur est appelé, un nouvel objet est crée, même si cela ne se passe pas à l'occasion de la définition d'une variable.

Par exemple, deux objets sans nom, représentant les points (1,2) et (3,4), sont créés dans l'instruction suivante :

```
    cout  << Point(1, 2).distance(Point(3, 4)) << endl;
```

3.4.4.Constructeur par défaut

Le constructeur par défaut est un constructeur qui peut être appelé sans paramètres.

Il est appelé dès qu'un objet est créé sans qu'il y ait appel explicite d'un constructeur, soit que le programmeur ne le juge pas utile, soit qu'il n'en a pas la possibilité :

```
    Point x;                          // équivaut à : Point x = Point ()
    Point t [10] ;                    // produit 10 appels de Point ()
    Point *p = new Point;             // équivaut à : p = new Point()
    Point *q = new Point[10];         // produit 10 appels de Point ()
```

Attention: L'utilisation de tableaux d'objets nécessite un constructeur par défaut.

Puisque tout objet doit être initialisé lors de sa création, si la classe ne comporte aucun constructeur, alors le compilateur synthétise un constructeur par défaut comme ceci :

- si la classe n'a ni objet membre, ni fonction virtuelle, ni classe de base (c'est le cas de notre classe Point), alors le constructeur synthétisé est le constructeur par défaut, qui consiste à ne rien faire. Les données membres seront créées comme elles l'auraient été en C, c'est-à-dire initialisées à zéro s'il s'agit de variables globales, laissées indéterminées s'il s'agit de variables locales ou dynamiques,
- si la classe a des objets membres ou des classes de base, alors le constructeur synthétisé produit l'appel du constructeur par défaut de chaque objet membre et de chaque classe de base.

Si au moins un constructeur est défini pour une classe, alors aucun constructeur par défaut n'est synthétisé par le compilateur. Par conséquent, ou bien l'un des constructeurs explicitement définis est un constructeur par défaut, ou bien toute création d'un objet devra expliciter des valeurs d'initialisation.

En C++, la notion de constructeur par défaut, simple en apparence, peut donner lieu à de multiples sources d'erreur, c'est pourquoi il convient de bien comprendre cette

notion.

3.4.5.Destructeurs

Le destructeur d'une classe est appelé lorsqu'un objet de la classe est détruit, juste avant que la mémoire occupée par l'objet soit récupérée par le système.

Un destructeur est une fonction membre particulière qui :

- a le même nom que la classe précédé du tilde : ~,
- n'a pas de paramètre,
- n'indique pas de type de retour,

Il y a donc au plus un destructeur par classe.

A titre d'exemple le programme suivant introduit une nouvelle variété de points : un nom (chaîne de caractères) est associé à chaque point.

Voici, par exemple, le destructeur pour la classe PointNomme :

```
class PointNomme
{
private:
    int x;
    int y;
    char* nom;
public:
    PointNomme(int abs, int ord, char *s = "")
    {
        x = abs;
        y = ord;
        nom = new char[strlen(s) + 1];
        strcpy(nom, s);
    }
    ~PointNomme()
    {
    delete[] nom ;
    }
};
```

Le destructeur assure la libération de l'espace alloué au point pour son nom.

Le destructeur est appelé différemment selon que l'objet auquel il appartient a été créé de façon statique ou dynamique.

- le destructeur d'un objet créé de façon statique est appelé de façon implicite dès que le programme quitte la portée dans lequel l'objet existe
- le destructeur d'un objet créé de façon dynamique doit être appelé grâce au mot clé *delete*, qui permet de libérer la mémoire occupée par l'objet.

Lorsque aucun destructeur n'a été défini, le compilateur en génère un, de la façon

suivante :

- si la classe n'a ni objets membres ni classes de base, alors il s'agit du destructeur par défaut qui consiste à ne rien faire,
- si la classe a des classes de base ou des objets membres, le destructeur généré consiste à appeler les destructeurs des données membres et des classes de base, dans l'ordre inverse de l'appel des constructeurs correspondants.

3.5.Exploitation des classes

3.5.1.Définition séparée et opérateur de résolution de portée

Tous les membres d'une classe doivent être au moins déclarés à l'intérieur de l'expression

```
class nom
{
…
};
```

qui constitue la déclaration de la classe.

Cependant, dans le cas des fonctions, aussi bien publiques que privées, on peut se limiter à n'écrire que leur en-tête à l'intérieur de la classe et définir le corps ailleurs, plus loin dans le même fichier ou bien dans un autre fichier.

Il faut alors un moyen pour indiquer qu'une définition de fonction, écrite en dehors de toute classe, est en réalité la définition d'une fonction membre d'une classe.

Ce moyen est fourni par l'opérateur de résolution de portée, dont la syntaxe est

```
NomDeClasse :: nomDeMethode.
```

Dans l'exemple de la classe Point on peut définir la méthode distance en dehors de la déclaration de la classe.

```
class Point
{
…
public:
      double distance(Point autrePoint);
…
} ;
```

La définition de la méthode distance est donnée plus loin ou dans un autre fichier de la manière suivante :

```
double Point ::distance(Point autrePoint)
{
      int dx = x – autrePoint.x;
      int dy = y – autrePoint.y;
      return sqrt(double(dx * dx + dy * dy));
```

```
}
```

Il est important de remarquer que les fonctions définies à l'intérieur d'une classe sont implicitement qualifiées en ligne.

La plupart des fonctions membres seront donc définies séparément. Seules les fonctions courtes, rapides et fréquemment appelées mériteront d'être définies dans la classe.

3.5.2.Fichier d'en-tête et fichier d'implémentation

Une partie importante de la programmation orientée objet consiste à définir des classes.

Généralement, ces classes sont utilisées par plusieurs programmes. Une programmation efficace doit organiser le code de manière à faciliter l'utilisation de ces différentes classes.

Voici comment on procède généralement :

- les définitions des classes se trouvent dans des fichiers en-tête (fichiers « *.h »),
- chacun des ces fichiers en-tête contient la définition d'une seule classe ou d'un groupe de classes intimement liées; par exemple, la définition de notre classe Point pourrait constituer un fichier point.h,
- les définitions des fonctions membres qui ne sont pas définies à l'intérieur de leurs classes sont écrites dans des fichiers sources (fichiers «*.cpp »),
- aux programmeurs, utilisateurs de ces classes sont distribués :
  - les fichiers «*.h »
  - les fichiers objets résultant de la compilation des fichiers «*.cpp »

Par exemple, voici les fichiers correspondants à notre classe Point :

```
////////////////////////////////////////////////// Fichier point.h ////////////////////////////////////////////////////////
#ifndef POINT_H
#define POINT_H
   class Point
   {
        int x, y ;
   public:
        void placer(int abs, int ord)                    //implicitement en ligne
        {
             x = abs;
             y = ord;
        }
        double distance(Point autrePoint);
   } ;
```

```
#endif
////////////////////////////////////////////////////// Fichier point. cpp ////////////////////////////////////////////////////////
   #include  "point.h"
   #include  <math.h>

   double Point ::distance(Point autrePoint)
   {
        int dx = x – autrePoint.x;
        int dy = y – autrePoint.y;
        return sqrt(double(dx * dx + dy * dy));
   }
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

La compilation du fichier point.cpp produira un fichier objet (nommé généralement point.o ou point.obj). Dans ces conditions, la distribution de la classe Point sera composée des deux fichiers point.h et point.obj, ce dernier ayant éventuellement été transformé en un fichier bibliothèque (nommé alors point.lib par exemple).

Tout programme utilisateur de la classe Point devra comporter la directive

```
#include "point.h"
```

et devra, une fois compilé, être relié au fichier point.obj ou point.lib.

3.6.Amis

3.6.1.Fonctions amies indépendantes

Une fonction qui, sans être membre d'une classe, a le droit d'accéder à *tous les membres* de cette classe, aussi bien publics que privés est une fonction amie de cette classe.

La déclaration ou la définition d'une fonction amie est écrite dans la classe qui accorde le droit d'accès à l'aide du mot réservé friend.

Cette déclaration doit être écrite indifféremment parmi les membres publics ou parmi les membres privés :

```
class Point
{
private:
     int x, y;
public:
     Point(int abs = 0, int ord = 0)
     {
          x = abs ;
          y = ord ;
     }
```

```
    friend int coincide(Point p, Point q) ;
};
int coincide(Point p, Point q)
{
    if (p.x == q.x && p.y == q.y) return 1;
    return 0 ;
}
```

Bien que déclarée à l'intérieur de la classe Point, *la fonction coincide n'est pas membre* de cette classe; en particulier, elle n'est pas attachée à un objet, et le pointeur this n'y est pas défini.

3.6.2.Fonction membre d'une classe, amie d'une autre classe

Il suffit dans ce cas de préciser, dans la déclaration d'amitié, la classe à laquelle appartient la fonction concernée.

Exemple :

```
class A
{
…
friend int B ::f(int x,A a);
…
};
class B
{
…
int f(int x,A a);
…
} ;

int B::f(int x,A a)
{
…                    /*f peut accéder aux membres privés de type A*/
}
```

3.6.3.Classes amies

Une classe amie d'une classe C est une classe qui a le droit d'accéder à tous les membres de C.

Une telle classe doit être déclarée dans la classe C (la classe qui accorde le droit d'accès), précédée du mot réservé friend, indifféremment parmi les membres privés ou parmi les membres publics de C.

Exemple :

```
class B ;                                   //B doit être connue de A
class A
{
…
friend class B;
…
};
class B
{
…

…
};
```

La relation d'amitié n'est pas transitive.

3.7.Constructeur de copie

Lorsqu'un objet est crée, C++ garantit qu'un constructeur, qui place cet objet dans un état initial acceptable, est appelé.

Dans certains cas, il est nécessaire de construire un objet même si le programmeur n'a pas prévu de constructeur à cette fin.

Le constructeur de copie est appelé lorsqu'un objet est initialisé en copiant un objet existant.

Le but du constructeur de copie est donc d'initialiser un objet lors de son instanciation à partir d'un autre objet.

Les constructeurs de copie sont appelés dans les cas suivants :

- une instance de classe est passée par valeur à une fonction,
- une instance de classe est retournée par une fonction,
- une instance de classe est initialisée avec une autre instance (Point a=b ;),
- une instance de classe est passée explicitement comme seul paramètre au constructeur (Point a(b) ;).

Il faut noter que dans les deux derniers cas cités plus haut, aucun appel au constructeur par défaut n'est effectué.

Le constructeur de copie d'une classe C est un constructeur dont le premier paramètre est de type C& (référence sur un C) ou const C& (référence sur un C constant) et dont les autres paramètres, s'ils existent, ont des valeurs par défaut.

Si aucun constructeur de copie n'a été défini pour une classe, un constructeur de copie

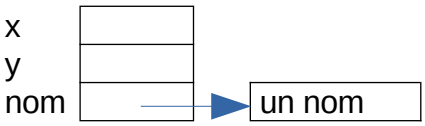
par défaut (différent du constructeur par défaut) est généré automatiquement par le compilateur. Ce constructeur se limite à la copie de chaque membre d'un objet dans le membre correspondant de l'autre objet. Il s'agit donc d'une copie superficielle.

Pour les objets contenant des pointeurs seules les valeurs des pointeurs sont recopiées (cela revient à faire la copie « bit à bit » d'un objet sur l'autre).

A titre d'exemple le programme suivant introduit une nouvelle variété de point ; à chacun est associée une étiquette qui est une chaîne de caractères :

```
class PointNomme
{
private:
    int x;
    int y;
    char*nom ;
public:
    PointNomme(int abs, int ord, char *s = "")
    {
        x = abs;
        y = ord;
        nom = new char[strlen(s) + 1];
        strcpy(nom, s);
    }
};
```

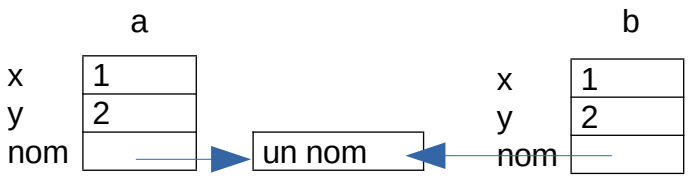
Lorsqu'un objet comporte des pointeurs, l'information qu'il représente ne se trouve pas entièrement incluse dans l'espace contigu que le compilateur connaît, car des morceaux d'information se trouvent à d'autres endroits de la mémoire.



La copie bit à bit que fait le compilateur peut être inadaptée à de tels objets. Si a et b sont deux objets de type PointNomme, une affectation

```
b = a;
```

engendrerait une copie bit à bit schématisée comme suit :



Ce type de copie est appelé copie superficielle d'un objet.

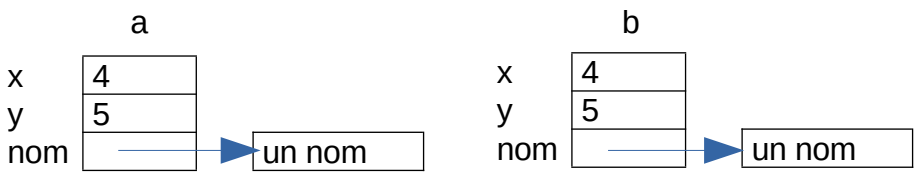
La copie du pointeur n'a pas dupliqué la chaîne pointée : les deux objets, l'original et la copie, partagent la même chaîne.

La plupart du temps ce partage n'est pas souhaitable (toute modification du nom d'un des deux points se répercutera immédiatement sur l'autre) il est même risqué (double libération d'un emplacement mémoire).

Le constructeur de copie permet de résoudre ce problème.

```
class PointNomme
{
private:
      int x;
      int y;
      char*nom ;
public:
      PointNomme(int abs, int ord, char *s = "")
      {
            x = abs;
            y = ord;
            nom = new char[strlen(s) + 1];
            strcpy(nom, s);
      }
      PointNomme(const PointNomme &p)          //constructeur de copie
      {
            x=p.x;
            y=p.y;
            nom = new char[strlen(p.nom) + 1];
            strcpy(nom,p.nom);
      }
};
```

La copie d'un point entraînera la duplication effective de la chaîne de caractères pointée :



Ce type de copie est appelé copie profonde d'un objet.

La définition d'un constructeur de copie pour une classe implique généralement que le constructeur de copie, le destructeur et l'opérateur d'affectation fournis par défaut par le compilateur ne conviennent pas pour cette classe.

Par conséquent, ces méthodes devront systématiquement être définies toutes les trois dès que l'une d'entre elle le sera.

3.8.Objets membres

Lorsque des membres d'une classe sont à leur tour d'un type classe on dit que la classe a des objets membres.

L'initialisation d'un objet de la classe nécessite alors l'initialisation de ces objets membres. Il en est toujours ainsi, indépendamment du fait que l'on emploie ou non un constructeur explicite, et qu'à son tour ce constructeur appelle ou non explicitement des constructeurs des objets membres.

Lorsque les objets membres n'ont pas de constructeurs par défaut, une syntaxe spéciale permet de préciser les arguments des constructeurs des membres :

```
NomDeLaClasse (paramètres)
                : membre (paramètres), . membre (paramètres)
        {
        corps du constructeur
        }
```

A titre d'exemple, imaginons que notre classe Point ne possède pas de constructeur sans arguments, et qu'on doive définir une classe Segment ayant deux points pour membres (un segment est déterminé par deux points).

Voici comment on devra écrire son constructeur :

```
class Segment
{
    Point origine, extremite;
    int couleur ;
public:
    Segment(int x1, int y1, int x2, int y2, int col) : origine(x1, y1), extremite(x2, y2)
    {
        couleur = col;
    }
} ;
```

La syntaxe spéciale pour l'initialisation des objets membres peut être utilisée aussi pour initialiser les données membres de types primitifs.

Par exemple, le constructeur de Segment précédent peut aussi s'écrire :

```
class Segment
{
…
public :
    Segment(int x1, int y1, int x2, int y2, int col)
        : origine(x1, y1), extremite(x2, y2), couleur(col) {…}
…
```

```
} ;
```

3.9.Agrégation

3.9.1.Agrégation interne - composition

Dans le cas de l'agrégation interne l'objet agrégé (par exemple point) est créé dans le constructeur de la classe qui l'agrège (par exemple figure). La responsabilité de sa destruction incombe également à la classe qui le contient (par exemple figure).

A titre d'exemple, nous allons agréger un objet de type Point dans chaque instance de la classe Figure.

```
class Figure
{
private:
    Point pointBase;
    int couleur;
    int epaisseur;
public:
    Figure (int x, int y, int v_couleur, int v_epaisseur);
...
};


Figure::Figure (int x, int y, int v_couleur, int v_epaisseur) :
     pointBase(x,y), couleur(v_couleur), epaisseur(v_epaisseur) {}
```

Le constructeur de la classe Figure commence par initialiser l'objet pointBase en appelant son constructeur avant d'initialiser les données membres couleur et epaisseur.

Comme nous l'avons vu au paragraphe précédent, il est très important de remarquer qu'il est *obligatoire* d'initialiser les données membres objets dans la liste d'initialisation du constructeur à moins que l'objet agrégé ne dispose d'un constructeur par défaut, auquel cas, *celui-ci sera appelé avant l'exécution du corps du constructeur*.

S'il n'y a pas de constructeur par défaut, il faudra recourir à un pointeur :

```
class Figure
{
private:
    int couleur;
    int epaisseur;
    Point *pointBase;
public:
```

```
    Figure (int x, int y, int v_couleur, int v_epaisseur);
    ~Figure()
    {
        delete pointBase;
    }
};


Figure::Figure (int x, int y, int v_couleur, int v_epaisseur) :
    couleur(v_couleur), epaisseur(v_epaisseur)
{
    pointBase=new Point(x,y);
}
```

3.9.2. Agrégation externe

A la différence de l'agrégation interne où l'objet agrégé est créé par l'objet agrégateur, l'agrégation externe repose sur l'utilisation d'un objet en provenance de l'extérieur.

Généralement, c'est le créateur de l'objet qui est responsable de sa destruction.

L'objet en provenance de l'extérieur ne sera donc généralement pas détruit par la classe qui l'agrège.

Il existe deux possibilités : agréger par référence ou par pointeur.

Dans l'agrégation par référence, la référence doit impérativement être initialisée dans la liste d'initialisation :

```
class Figure
{
private:
    int epaisseur;
    int couleur;
    Point& pointBase;
public:
    Figure (Point &p, int v_couleur, int v_epaisseur) : pointBase(p)
    {
            couleur=v_couleur;
            epaisseur=v_epaisseur;
    }
...
};
```

Dans le cas de l'agrégation par pointeur, nous avons :

```
class Figure
{
private:
```

```
    int epaisseur;
     int couleur;
    Point *pointBase;
public:
    Figure(Point &p, int v_couleur, int v_epaisseur)
    {
            couleur=v_couleur;
            epaisseur=v_epaisseur;
            pointBase=&p;
    }
    ...
};
```

3.9.3.Accès à l'objet agrégé

Pour donner accès en lecture seule à l'objet agrégé le plus simple est de renvoyer une référence constante.

Agrégation interne :

```
const Point& Figure::pointDeBase() const
{
    return pointBase;
}
```

Agrégation externe par référence :

```
const Point& Figure::pointDeBase() const
{
    return pointBase;
}
```

Agrégation externe par pointeur :

```
const Point& Figure::pointDeBase() const
{
    return *pointBase;
}
```

3.10.Membres constants

3.10.1.Données membres constantes

Une donnée membre d'une classe peut être qualifiée const. Il est alors obligatoire de l'initialiser lors de la construction d'un objet, et sa valeur ne pourra par la suite plus être modifiée.

A titre d'exemple voici une nouvelle version de la classe Segment, dans laquelle chaque objet reçoit, lors de sa création, un « numéro » qui ne doit plus changer au cours de la vie de l'objet :

```
class Segment
{
    Point origine, extremite;
    int couleur;
    const int numero;                          //numero est constant
public :
    Segment(int x1, int y1, int x2, int y2, int col, int num);
} ;
```

Constructeur, en utilisant la syntaxe de l'initialisation des objets membres :

```
Segment::Segment(int x1, int y1, int x2, int y2, int col, int num)
        :origine(x1, y1), extremite(x2, y2), couleur(col), numero(num){}
```

La donnée membre constante est donc initialisée dans la liste d'initialisation.

3.10.2.Fonctions membres constantes

Le mot const placé à la fin de l'en-tête d'une fonction *membre* indique que l'état de l'objet à travers lequel la fonction est appelée n'est pas changé du fait de l'appel.

C'est une manière de déclarer qu'il s'agit d'une fonction de consultation de l'objet, non d'une fonction de modification :

```
class Point
{
…
    void placer(int a, int b);                 // modifie l'objet
    float distance (Point p) const;            // ne modifie pas l'objet
} ;
```

Cela permet au compilateur d'autoriser certains accès qui, sans cela, auraient été interdits.

Exemple :

```
void uneFonction(const Point a)
{
    Point b;
    double d = a.distance(b);
}
```

la qualification const de la fonction distance est indispensable pour que l'expression précédente soit acceptée par le compilateur. C'est elle seule, en effet, qui garantit que le point a, contraint à rester constant, ne sera pas modifié par l'appel de distance.

Il est conseillé de qualifier const toute fonction qui peut l'être : comme l'exemple précédent le montre, cela élargit son champ d'application.

La qualification const d'une fonction membre fait partie de sa signature. Ainsi, on peut surcharger une fonction membre non constante par une fonction membre constante ayant, à part cela, le même en-tête. La fonction non constante sera appelée sur les

objets non constants, la fonction constante sur les objets constants.

Les fonctions membres statiques n'admettent pas la qualification const.

On peut utiliser cette propriété pour écrire des fonctions qui n'effectuent pas le même traitement ou qui ne rendent pas le même type de résultat lorsqu'elles sont appelées sur un objet constant et lorsqu'elles sont appelées sur un objet non constant.

Exemple :

```
class Point
{
      int x, y;
public :
      …
      int get_x() const {return x;}
      int get_y() const {return y;}
      int& get_x() {return x;}                    //surcharge autorisée
      int& get_y() {return y;}
};
```

Avec la déclaration précédente, les fonctions get_x et get_y sont sécurisées : sur un objet constant elles ne permettent que la consultation, sur un objet non constant elles permettent la consultation et la modification :

```
const Point a(1, 2);

Point b(3,4);

int r;

r = a.get_x();                    // OK

a.get_x()  = r;                   // ERREUR a est constant

r = b.get_x();                    // OK

b.get_x() = r;                    // OK
```

La notion de fonction membre constante n’existe pas en Java.

3.10.3.Membres mutables

Une fonction membre constante ne peut modifier les valeurs des membres non statiques.

Le qualificatif mutable placé devant une donnée membre permet à une fonction membre constante de modifier cette donnée.

Exemple :

```
class Point
{
      mutable int x ;
      …
```

```
        void translater() const
        {
            x++;                                        //permis car x est mutable
        }
        …
};
```

4.Surcharge des opérateurs

4.1.Principe

Nous avons vu précédemment qu'il était possible de surcharger une fonction.

En C++, (contrairement au JAVA) il est également possible de surcharger un opérateur.

Surcharger un opérateur consiste à redéfinir la signification d'un opérateur existant pour l'étendre à des objets ou pour changer l'effet d'opérateurs prédéfinis sur des objets.

Cette technique très puissante permet de créer, par l'intermédiaire des classes, des types à part entière munies d'opérateurs intégrés tout comme les types de base.

Seuls les opérateurs déjà connus du compilateur peuvent être surchargés.

Tous les opérateurs de C++ peuvent être surchargés, sauf les suivants :

:: . * ? : sizeof typeid static_cast dynamic_cast const_cast reinterpret_cast

Pour surcharger un opérateur, un opérande au moins doit être du type classe.

Une fois surchargés, les opérateurs gardent leur pluralité, leur priorité et leur associativité initiales.

En revanche, ils perdent leur éventuelle commutativité et les éventuels liens sémantiques avec d'autres opérateurs.

Par exemple, la sémantique d'une surcharge de ++ ou <= n'a pas à être liée avec celle de + ou <.

Surcharger un opérateur revient à définir une fonction; tout ce qui a été dit à propos de la surcharge des fonctions s'applique donc à la surcharge des opérateurs.

Plus précisément, pour surcharger un opérateur ⊗ (le signe ⊗ représente un opérateur quelconque) il faut définir une fonction nommée operator⊗.

Ce peut être une fonction membre d'une classe ou bien une fonction indépendante. Si elle n'est pas membre d'une classe, alors elle doit avoir au moins un paramètre d'un type classe.

4.1.1.Surcharge d'un opérateur par une fonction membre

Si la fonction operator⊗ est membre d'une classe, elle doit comporter un paramètre de moins que la pluralité de l'opérateur: le premier opérande sera l'objet à travers lequel la fonction a été appelée.

Ainsi, à quelques exceptions près:

- obj⊗ ou ⊗obj sont équivalents à obj.operator⊗()

- obj1⊗obj2 est équivalent à obj1.operator⊗(obj2)

Exemple :

```
class Point
{
    int x, y;
public:
    Point(int = 0, int = 0);
    int get_x() const { return x; …}
    int get_y() const { return y;…}
    Point operator+(const Point) const;        // surcharge de + par une fonction
                                               // membre
…
} ;


Point Point ::operator+(const Point q) const
{
    return Point(x + q.x, y + q.y);
}
```

Utilisation:

```
Point p, q, r;
r = p + q;                                     //équivalent à : r = p.operator+(q) ;
```

Les opérateurs =, [ ], () et → ne peuvent être surchargés que par des fonctions membres.

4.1.2.Surcharge d'un opérateur par une fonction non membre

Si la fonction operator⊗ n'est pas membre d'une classe, alors elle doit avoir un nombre de paramètres égal à la pluralité de l'opérateur.

Dans ce cas :

- obj⊗ ou ⊗obj sont équivalents à operator⊗ (obj)
- obj1⊗obj2 est équivalent à operator⊗ (obj1, obj2)

Exemple :

```
Point operator+(const Point p, const Point q)
{
    return Point (p.get_x() + q.get_x(), p.get_y() + q.get_y()) ;
                                               // surcharge de + par une
}                                              // fonction non membre
```

Utilisation :

```
Point p, q, r;
r = p + q;                          //équivalent à : r = operator+(p, q) ;
```

A cause des conversions implicites, la surcharge d'un opérateur binaire symétrique par une fonction non membre, comme la précédente, est en général préférable, car les deux opérandes y sont traités symétriquement.

Exemple :

```
Point p, q, r;
int x, y;
```

Surcharge de l'opérateur + par une fonction membre :

```
r = p + q;                          // r = p.operator+(q);
r = p + y;                          // r = p.operator+(Point(y));
r = x + q;                          // Erreur : x n'est pas un objet
```

Surcharge de l'opérateur + par une fonction non membre :

```
r = p + q;                          // r = operator+(p, q);
r = p + y;                          // r = operator+(p, Point(y));
r = x + q;                          // r = operator+(Point(p), q);
```

Lorsque la surcharge d'un opérateur n'est pas une fonction membre, il est alors souvent judicieux d'en faire une fonction amie.

Si, par exemple, la classe Point n'avait pas possédé les accesseurs publics get_x() et get_y(), on aurait dû surcharger l'addition par une fonction amie :

```
class Point
{
    int x, y;
public:
    Point(int = 0, int = 0);
    friend Point operator+(const Point, const Point);
};


Point operator+(const Point p, Point q)
{
    return Point(p.x + q.x, p.y + q.y);
}
```

Les deux surcharges de l'opérateur +, par une fonction membre et par une fonction non membre, ne peuvent pas être définies en même temps dans un même programme puisqu'une expression comme p + q serait trouvée ambiguë par le compilateur.

On notera que cela est une particularité de la surcharge des opérateurs, un tel

problème ne se pose pas pour les fonctions ordinaires (une fonction membre n'est jamais en compétition avec une fonction non membre).

La surcharge d'un opérateur binaire par une fonction non membre est obligatoire lorsque le premier opérande est d'un type standard ou d'un type classe défini par ailleurs, que le programmeur ne peut plus étendre.

4.2.Exemples

4.2.1.Injection et extraction de données dans les flux

L'opérateur << peut être utilisé pour injecter des données dans un flux de sortie, ou ostream.

La classe ostream a défini des surcharges de << :

```
class ostream
{
public:
    ostream& operator<<(int) ; ostream& operator<<(unsigned int) ;
    ostream& operator<<(long); ostream& operator<<(unsigned long);
    ostream& operator<<(double) ; ostream& operator<<(long double) ;
    etc.
} ;
```

Il n'est pas possible d'ajouter des membres à la classe ostream, si l'on souhaite étendre << aux objets d'une classe, il faut donc écrire une fonction non membre amie ou non de la classe:

```
ostream& operator<<(ostream& o, const Point p)
{
    return o << "(" << p.get_x() << "," << p.get_y() <<  ")";
}
```

L'opérateur >> permet d'extraire des données à partir de flux de sortie ou istream.

La classe istream a défini des surcharges de >> :

```
class istream
{
public:
    istream& operator>>(int) ; istream& operator>>(unsigned int) ;
    istream& operator>>(long); istream& operator>>(unsigned long);
    istream& operator>>(double) ; istream& operator>>(long double) ;
    etc.
} ;
```

Comme pour l'opérateur <<, pour étendre l'opérateur >> aux objets d'une classe il faut écrire une fonction non membre, amie ou non de la classe :

```
istream& operator >> (istream& i, Point& p)
{
    cout <<"Abscisse:"; i >> p.x;
    cout << "Ordonnée:"; i >> p.y;
    return i ;
}
```

Exemple :

```
class Point
{
    ...
                                                    // avec une fonction amie
    friend istream& operator>>(istream&, Point&);
...
};

istream& operator>>(istream& i, Point& p)
{
    cout << "x: " ;i >> p.x;
    cout << "y: "; i >> p.y;
    return i ;
}
ostream& operator<<(ostream& o, Point p)        // sans relation d'amitié
{
    o << "x: " << get_x() << "y: " << get_y();
    return o;
}
```

Utilisation :

```
Point p1,p2;
cin >> p1 >> p2 ;
cout  << "Les point se trouvent en :"  << p1 << p2 << "\n";
```

4.2.2.Affectation

En l'absence de surcharge explicite, il n'y a aucun appel au constructeur de copie, l'opérateur d'affectation correspond à une copie des valeurs de son second opérande dans le premier.

Comme nous l'avons constaté, cette simple copie peut s'avérer insuffisante dès lors que les objets concernés comportent des pointeurs sur des emplacements dynamiques. Il s'agit là typiquement d'une situation le situation qui demande la surcharge de l'opérateur =.

L'opérateur d'affectation doit être surchargé par une fonction membre (en effet, dans une affectation on ne souhaite pas que les opérandes jouent des rôles symétriques).

Exemple :

```
class PointNomme
{
      int x, y ;
      char *nom ;
public:
      PointNomme(int = 0, int = 0, char * = "");
      PointNomme(const PointNomme&);
      ~PointNomme();
} ;

      PointNomme::PointNomme(int abs, int ord, char *s)
      {
            x = abs;
            y = ord;
            nom = new char[strlen(s) + 1];
            strcpy(nom, s);
      }

PointNomme::PointNomme(const PointNomme& p)
{
      cout << "PointNomme(const PointNomme&) << endl;
      x = p.x;
      y = p.y;
      nom = new char[strlen(p.nom) + 1];
      strcpy(nom,  p.nom);
}

PointNomme::~PointNomme()
{
      cout <<  "~PointNomme() <<endl;
      delete [ ] nom;
}
```

Voici un exemple d'utilisation de la classe PointNomme aux conséquences fatales :

```
void main()
{
      PointNomme p(0, 0, "Origine"), q;
      q = p;
}
```

L'opérateur = n'ayant pas été surchargé, l'instruction q = p ne fait qu'une copie bit à bit de l'objet p dans l'objet q, c'est à dire une copie superficielle.

A la fin du programme, les objets p et q sont détruits, l'un après l'autre. La destruction d'un de ces objets libère la chaîne nom et rend l'autre objet incohérent, ce qui provoque une erreur fatale lors de la restitution du second objet.

Pour résoudre le problème, on surcharge l'opérateur = de manière à assurer la destruction de la valeur courante suivie d'une copie :

```
class PointNomme
{
 int x, y ;
 char *nom ;
public:
      PointNomme(int = 0, int = 0, char * = "");
      PointNomme(const PointNomme&) ;
      PointNomme& operator=(const PointNomme&);
      ~PointNomme();
} ;

PointNomme& PointNomme::operator=(const PointNomme& p)
{
      if (&p != this)
      {
            delete [ ] nom;
            x = p.x;
            y = p.y;
            nom = new char[strlen(p.nom) + 1];
            strcpy(nom, p.nom);
      }
      return *this;
}
```

4.3.Forme canonique de Coplien

Une classe T est sous forme canonique de Coplien lorsqu'elle fournit les éléments suivants :

Prototype	Fonctionnalité
T::T()	Constructeur par défaut
T::T(const T&)	Constructeur de copie
T& T::operator=(const T&)	Opérateur d'affectation
T::~T()	Destructeur

L'utilisation d'une classe sous forme canonique permet une gestion sécurisée de la mémoire. La forme canonique s'impose lorsqu'une classe utilise de la mémoire

dynamique ou des ressources critiques.

4.4.Opérateurs de conversion

4.4.1. Conversion vers un type classe

Le constructeur de conversion permet de convertir une donnée de n'importe quel type, primitif ou classe, vers un type classe.

Ce constructeur est un constructeur qui peut être appelé avec un seul argument.

Exemple :

```
class Point
{
public:
      Point(int a)
      {
            x = y = a;
      }
} ;
```

Ce constructeur peut être appelé de trois manières différentes :

```
Point p = Point(1);
Point q(2) ;                                  //  Point q = Point (2);
Point r = 3;                                  //  Point r = Point(3);
```

La troisième expression ci-dessus fait bien apparaître l'aspect conversion de ce procédé. Il faut savoir qu'un tel constructeur sera aussi appelé implicitement, car le compilateur s'en servira à chaque endroit où, un Point étant requis, il trouvera un nombre:

```
Point p;
r = p + 5;                                    //  r = operator+(p, Point(5));
```

Si on les trouve trop dangereuses, on peut empêcher que le compilateur fasse de telles utilisations implicites d'un constructeur à un argument. Il suffit pour cela de le qualifier explicit :

```
class Point
{
public:
      explicit Point(int a)
      {
            x = y = a;
      }
} ;
```

Utilisation :

```
Point p = Point(1);                    // Ok
Point q = 2;                           // Erreur
```

4.4.2.Conversion d'un objet vers un type primitif ou non vers un autre objet

C étant une classe et T un type, primitif ou non, on définit la conversion de C vers T par une fonction membre C::operatorT().

Par exemple, les classes Point et Segment suivantes sont munies de conversions int ↔ Point et Point ↔ Segment :

```
class Point
{
…
    int x, y;
    Point (int a)                      // conversion int → Point
    {
        x = a;
        y = 0;
    }
    operator int()                     // conversion Point → int
    {
        return abs(x) + abs(y);
    };


class Segment
{
…
    Point p1, p2;
    Segment (Point p): p1(p), p2(p) { }    // conversion Point →
Segment
    operator Point()                   // conversion Segment
→Point
    {
        return Point((p1.x + p2.x)/2, (p1.y + p2.y)/2);
    }
```

5.Héritage

5.1.Classes de base et classes dérivées

L'héritage est à la base des possibilités de réutilisation de composants logiciels que sont les classes.

L'héritage peut être vu comme un outil de spécialisation croissante. Le mécanisme de l'héritage consiste en la définition d'une classe par réunion des membres d'une (héritage simple) ou plusieurs (héritage multiple) classes préexistantes, dites classes de base, et d'un ensemble de membres spécifiques de la classe nouvelle, appelée alors classe dérivée.

L'héritage n'est pas limité à un seul niveau : une classe dérivée peut devenir à son tour classe de base pour une autre classe.

C++ autorise l'héritage multiple : une classe être dérivée de plusieurs classes de base.

L'utilisation de l'héritage multiple ne fait pas l'unanimité parmi les programmeurs, elle ne sera pas abordée dans le cadre de ce cours.

La syntaxe est :

```
class classeDérivée : dérivation classeBase1, dérivation classeBase2, …
dérivation classe
{
déclarations et définitions des membres spécifiques de la nouvelle classe
}
```

où dérivation est un des mots-clés private, protected ou public.

Exemple : La classe pixel est un point amélioré qui comporte une information de type char sur sa couleur :

```
class Point
{
    int x, y;
public :
    Point(int, int);
void afficher()                                  // affichage d'un Point
{
    cout << " (" << x << ", "  << y << " ) " ;
}
} ;

class Pixel : public Point                       // Pixel dérive publiquement
                                                 // de Point
{
    char *couleur;
public :
    Pixel(int, int, char *);
```

```
    void afficher_pixel()                    // affichage d'un Pixel
    {
        cout << " [ " ;
        afficher() ;                         // affichage en tant que Point
        cout << ";" << couleur << " ] ";
    }
} ;
```

La déclaration

```
    class Pixel : public Point
```

précise que Pixel est une classe dérivée de la classe de base Point.

Le mot public signifie que les membres publics de la classe de base Point seront des membres publics de la classe Pixel.

Par défaut, une classe dérive de manière privée.

La méthode afficher() est donc accessible depuis la classe Pixel.

5.2.Redéfinition (overriding) d'une fonction membre

Au lieu d'utiliser la fonction afficher_pixel(), on peut redéfinir (override) la fonction afficher(), un objet de type Pixel appellera alors la redéfinition implémentée dans la classe Pixel.

Lorsqu'une fonction membre est redéfinie dans une classe dérivée, elle masque toutes les fonctions membres de même nom de la classe de base.

Le masquage d'une fonction membre de la classe de base par une fonction membre de même signature se justifie pleinement dans certaines situations.

Dans notre exemple, un Pixel est un point amélioré (augmenté d'un membre supplémentaire, sa couleur).

L'affichage d'un pixel consiste à l'afficher en tant que point, avec des informations additionnelles.

Pour la classe Pixel, nous avons:

```
class Point
{
    int x, y;
public :
    Point(int, int);
    void afficher()
    {
        cout << " (" << x << ", "  << y << " ) " ;
    }
} ;

class Pixel : public Point
{
    char *couleur;
public :
```

```
        Pixel(int, int, char *);
        void afficher()                          // redéfinition de la fonction
                                                 // afficher()
        {
            cout << " [ " ;
            Point :: afficher() ;                // opérateur :: nécessaire
            cout << ";" << couleur << " ] ";
        }
    } ;
```

Généralement, cet enrichissement permet aux fonctions redéfinies d'accéder à ce que la classe dérivée a de plus que la classe de base.

5.3.Création et destruction des objets dérivés

A l'occasion de la construction d'un objet, ses sous-objets hérités sont initialisés par l'intermédiaire des constructeurs des classes de base correspondantes.

Lorsque des arguments sont nécessaires, il faut les mentionner dans le constructeur de l'objet selon la syntaxe suivante:

```
classe (paramètres)
    : classeDeBase (paramètres), classeDeBase (paramètres)
    {
        corps du constructeur
    }
```

Les destructeurs des classes de base directes sont toujours appelés lors de la destruction d'un objet.

Cet appel est toujours implicite.

Exemple :

```
class Point
{
…
public :
    Point (int abs, int ord);
…
};

class Pixel : public Point
{
…
public:
    Pixel(int abs, int ord, char*c) : Point(abs,ord)  // Transmission au constructeur
                                                      // Point de abs et ord.
…
};
```

5.4.Contrôle des accès

C++ permet d'intervenir sur deux sortes d'autorisations d'accès aux membres de la classe de base :

- accès par les fonctions membres d'une classe dérivée,
- accès par les utilisateurs (=tout objet) de la classe dérivée.

En plus des membres publics et privés, une classe C peut avoir des membres protégés.

5.4.1.Membres protégés

Les membres protégés sont annoncés par le mot clé protected, ils présentent une accessibilité intermédiaire car ils sont accessibles par les fonctions membres et amies de C et aussi par les fonctions membres et amies des classes directement dérivées de C.

Les membres protégés sont donc des membres qui ne font pas partie de l'interface de la classe, mais dont on a jugé que le droit d'accès serait nécessaire ou utile aux concepteurs des classes dérivées.

Exemple :

```
class Point
{
protected :
      int x, y;
public :
      Point(int, int);
…
 } ;

class Pixel : public Point                          // Pixel dérive de Point
{
      char *couleur;
public :
      Pixel(int, int, char *);
      void afficher_pixel()                         // affichage d'un Pixel
      {
            cout << " [ " ;
            cout << " (" << x << ", "  << y << " ) " ;      // x et y sont accessibles aux
            cout  << ";" << couleur << "]";                 // méthodes de Pixel
      }
} ;
```

5.4.2.Dérivation privée, protégée, publique

Le choix de la dérivation par un mot clé parmi private, protected ou public, détermine

l'accessibilité dans la classe dérivée des membres de la classe de base.

Les objets des classes dérivées peuvent donc avoir des membres inaccessibles : les membres privés de la classe de base sont présents dans les objets de la classe dérivée, mais il n'y a aucun moyen d'y faire référence.

Héritage public:

```
class classeDérivée  :  public classeDeBase { ... }
```

C'est la forme d'héritage la plus fréquemment utilisée, car la plus utile et la plus facile à comprendre : dire que D dérive publiquement de B c'est dire que, vu de l'extérieur, tout D est une sorte de B, ou encore que tout ce qu'on peut demander à un B, on peut aussi le demander à un D.

HERITAGE PUBLIC			
Statut dans la classe de base	Accès par les fonctions membres et amies de la classe dérivée	Accès par un utilisateur de la classe dérivée	Nouveau statut dans la classe dérivée
public	Oui	Oui	public
protégé	Oui	Non	protégé
privé	Non	Non	privé

Héritage protégé:

```
class  classeDérivée : protected classeDeBase { ...  }
```

HERITAGE PROTEGE			
Statut dans la classe de base	Accès par les fonctions membres et amies de la classe dérivée	Accès par un utilisateur de la classe dérivée	Nouveau statut dans la classe dérivée
public	Oui	**Non**	**protégé**
protégé	Oui	Non	protégé
privé	Non	Non	inaccessible

Héritage privé:

```
class  classeDérivée : private classeDeBase { ... }
```

Le mot clé private est optionnel car l'héritage privé est l'héritage par défaut. C'est la forme la plus restrictive d'héritage.

Dans l'héritage privé, l'interface de la classe de base disparaît (l'ensemble des membres publics cessent d'être publics). Autrement dit, on utilise la classe de base pour réaliser l'implémentation de la classe dérivée, mais on s'oblige à écrire une nouvelle interface pour la classe dérivée.

HERITAGE PRIVE			
Statut dans la classe de base	Accès par les fonctions membres et amies de la classe dérivée	Accès par un utilisateur de la classe dérivée	Nouveau statut dans la classe dérivée
public	**Non**	Non	**privé**
protégé	**Non**	Non	**privé**
privé	Non	Non	inaccessible

Exemple :

```
class Point
{
      int x, y;
protected :
      deplacer() ;
public :
      Point(int, int);
      void afficher();
      void nommer() ;
…
 } ;

class Pixel : private Point
{
      char *couleur;
public :
      Pixel(int, int, char *);
      void afficher() ;
…
} ;
```

Utilisation :

```
Pixel px :
px.deplacer()                    // Erreur
px.nommer() ;                    // Erreur
px.afficher() ;                  // Appel de la méthode afficher
                                 // de Pixel
```

Si D dérive de manière privée de B alors un objet D est une sorte de B, mais les utilisateurs de ces classes n'ont pas à le savoir. Ou encore : ce qu'on peut demander à un B, on ne peut pas forcément le demander à un D.

En JAVA : alors que Java dispose des trois statuts public, protégé (avec une signification plus large qu'en C++) et privé, il ne dispose que d'un seul mode de dérivation correspondant à la dérivation publique du C++.

5.5.Synthèse sur la création et la destruction des objets

5.5.1.Construction

1. Le constructeur d'un objet est appelé directement après l'obtention de l'espace pour l'objet s'il n'est pas lui-même la valeur d'une variable globale.

   L'espace mémoire destiné aux variables globales existe dès que le programme est chargé. Les constructeurs de ces variables sont appelés, *dans l'ordre des déclarations de ces variables,* juste avant l'activation de la fonction principale du programme (en principe main).

2. L'activation d'un constructeur commence par les appels des constructeurs de chacune de ses classes de base directes, *dans l'ordre de déclaration de ces classes de base directes.*

3. Sont appelés ensuite les constructeurs de chacun des objets membres, *dans l'ordre de déclaration de ces objets membres.*

4. Enfin, les instructions qui composent le corps du constructeur sont exécutées.

Sauf indication contraire, les constructeurs dont il est question aux points 2 et 3 sont les constructeurs sans argument des classes de base et des classes des données membres. Si des arguments doivent être précisés (par exemple parce que certaines de ces classes n'ont pas de constructeur sans argument) cela se fait selon la syntaxe :

```
classe (parametres)
        : nomDeMembreOuDeClasseDeBase (paramètres),
        nomDeMembreOuDeClasseDeBase (paramètres)
        {
                corps du constructeur
        }
```

5.5.2.Destruction

Le principe général est celui-ci : les objets « contemporains » (attachés à un même contexte, crées par une même déclaration, etc.) sont détruits dans l'ordre inverse de leur création. Par conséquent

1. L'exécution du destructeur d'un objet débute par l'exécution des instructions qui en composent le corps.

2. Elle se poursuit par les appels des destructeurs des classes des objets membres, dans l'ordre inverse de leurs déclarations.

3. Les destructeurs des classes de base directes sont invoqués ensuite dans l'ordre inverse des déclarations de ces classes de base.

4. L'espace occupé par l'objet est libéré.

Les objets qui sont les valeurs de variables globales sont détruits immédiatement après la fin de la fonction principale (en principe main), dans l'ordre inverse de la déclaration des variables globales.

Les objets qui ont été alloués dynamiquement ne sont détruits que lors d'un appel de delete les concernant.

Les objets qui sont des valeurs de variables automatiques (locales) sont détruits lorsque le contrôle quitte le bloc auquel ces variables sont rattachées ou, au plus tard, lorsque la fonction contenant leur définition se termine.

Les objets temporaires ont les vies les plus courtes. En particulier, les objets temporaires créés lors de l'évaluation d'une expression sont détruits avant l'exécution de l'instruction suivant celle qui contient l'expression.

Les objets temporaires créés pour initialiser une référence persistent jusqu'à la destruction de cette dernière.

Exemple :

```
Point &r = Point (1, 2) ;                    // l'objet temporaire dure autant
                                             // que r,
                                             // au moins jusqu'à la fin du bloc
```

Attention, les pointeurs ne sont pas traités avec autant de soin. Il ne faut jamais initialiser un pointeur avec l'adresse d'un objet temporaire (l'opérateur new est fait pour cela) :

```
Point *p = & Point (1, 2);                   // Erreur
Point *q = new Point (1, 2);                 // OK
```

En l'absence de constructeur de copie, le pointeur p *est invalide : le point de coordonnées (1,2) a déjà été détruit*

```
delete q;                                    // OK
delete p;                                    // Erreur
```

N.B : Lorsqu'un constructeur de copie a été défini dans la classe de base, tout constructeur de copie défini dans une sous classe doit faire appel explicitement au constructeur de copie de la classe de base.

5.6.Polymorphisme

5.6.1.Conversion prédéfinie vers une classe de base

Si la classe *D* dérive publiquement de la classe *B* alors les membres de *B* sont membres de *D.* Autrement dit, les membres publics de *B* peuvent être atteints à travers un objet *D.* Ou encore : *tous les services offerts par un B sont offerts par un D.*

Par conséquent, là où un *B* est prévu, on doit pouvoir mettre un *D.* C'est la raison pour laquelle la conversion (explicite ou implicite) d'un objet de type *D* vers le type *B,* une classe de base accessible de *D,* est définie, et a le statut de **conversion standard**.

Exemple :

```
class Point
{
    int x, y;
public :
    Point(int, int);                    // point
} ;


class Pixel : public Point
{
    char *couleur;
public :
    Pixel(int, int, char *);
} ;
```

Utilisation :

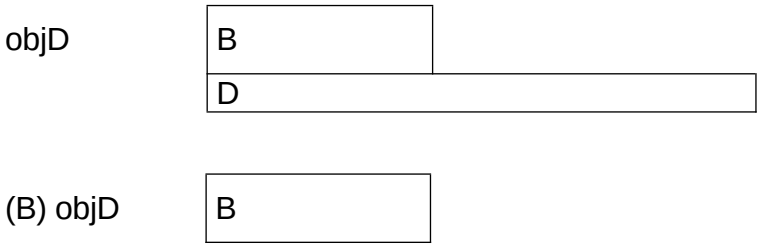
```
Pixel px(1, 2, "BLEU");
Point pt = px;                          // conversion implicite
                                        // d'un pixel en un point
                                        // « px n'a plus de couleur »
```

La conversion d'un *D* vers un *B* est traitée comme l'appel d'une fonction membre de *D* qui serait publique, protégée ou privée selon le mode dont *D* dérive de *B.*

Dans la suite de ce paragraphe, nous supposerons être dans le cas le plus courant d'une dérivation publique.

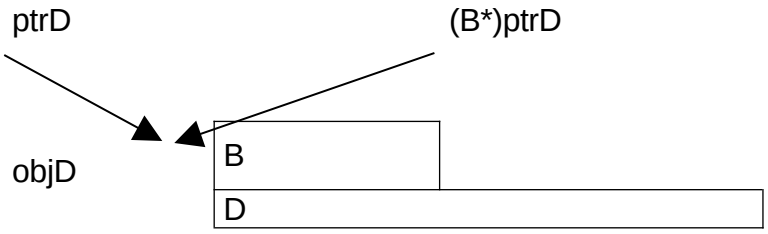
La conversion d'un objet de la classe dérivée vers la classe de base a des effets fort différents suivant qu'elle s'applique à des objets ou à des pointeurs (ou des références) sur des objets:

1. Convertir un objet *D* vers le type *B* c'est lui enlever tous les membres qui ne font pas partie de *B* (les membres spécifiques et ceux hérités d'autres classes). Dans cette conversion il y a perte effective d'information :



2. Convertir un *D** en un *B** (c'est-à-dire un pointeur sur *D* en un pointeur sur *B)* ne fait perdre aucune information.

   Pointé à travers une expression de type *B*,* l'objet n'offre que les services de la classe *B,* mais il ne cesse pas d'être un *D* avec tous ses membres :



Utilisation :

```
void f1(Point pt) ;
void f2(Point *pt) ;
void f3(Point &pt) ;
Pixel px = Pixel(10, 20, "BLEU")
f1(px);                    // px n'a plus de couleur
f2(&px);                   // px a encore une couleur
f3(px);                    // px a encore une couleur
```

L'objet passé à f1 comme argument est une version tronquée de px, alors que le pointeur passé à f2 est l'adresse de l'objet px tout entier, il en est de même pour la référence passée à f3.

5.6.2.Type statique, type dynamique

En C++, un pointeur sur un type d'objet peut recevoir l'adresse de n'importe quel objet descendant. Toutefois, ce mécanisme dispose d'une lacune importante : l'appel d'une méthode pour un objet pointé conduit systématiquement à appeler la méthode correspondant au *type du pointeur* et non pas au type effectif de l'objet pointé lui-même.

Cette lacune provient principalement de ce que C++ réalise un typage statique ou

encore une ligature statique.

Le type d'un objet (pointé) y est déterminé au moment de la compilation. Dans ces conditions, le mieux que puisse faire le compilateur est effectivement de considérer que l'objet pointé a le type du pointeur.

De manière à réaliser l'appel de la méthode correspondant au type de l'objet pointé, il est nécessaire que le type de l'objet ne soit pris en compte qu'au moment de l'exécution.

On parle alors de typage dynamique, de ligature dynamique ou encore de polymorphisme.

C++ le typage dynamique peut être mis en œuvre grâce au mécanisme des fonctions virtuelles.

Considérons la situation suivante :

```
class B
{
 …
};

class D : public B
{
…
};


D unD;
B unB = unD;
B *ptrB = &unD;
B &refB = unD;
```

Les trois affectations ci-dessus sont légitimes ; elles font jouer la conversion standard d'une classe vers une classe de base.

Le type de unB ne soulève aucune question (c'est un B), mais les choses sont moins claires pour les types de ptrB et ref B.

Ainsi ptrB, par exemple, pointe-t-il un B ou un D ?

- on dit que le type statique de *ptrB (ce que ptB pointe) et de ref B est B, car c'est ainsi que ptB et ref B ont été déclarées ;
- on dit que le type dynamique de *ptrB et de ref B est D, car tel est le type des valeurs effectives de ces variables.

Le type statique d'une expression découle de l'analyse du texte du programme ; il est

connu à la compilation.

Le type dynamique, au contraire, est déterminé par la valeur courante de l'expression, il peut changer durant l'exécution du programme.

5.7.Fonctions virtuelles

Soit f une fonction membre d'une classe *C.*

Si les conditions suivantes sont réunies :

- f est redéfinie dans des classes dérivées (directement ou indirectement) de C,
- f est souvent appelée à travers des pointeurs ou des références sur des objets de *C* ou de classes dérivées de C,

alors f mérite d'être une *fonction virtuelle.*

On exprime cela en faisant précéder sa déclaration du mot réservé virtual.

Si f est appelée à travers un pointeur ou une référence sur un objet de C, le choix de la fonction effectivement activée, parmi les diverses redéfinitions de f, se fera d'après le type dynamique de cet objet.

Une classe possédant des fonctions virtuelles est dite *classe polymorphe*.

La qualification virtual devant la redéfinition d'une fonction virtuelle est facultative : les redéfinitions d'une fonction virtuelle sont virtuelles d'office.

Exemple :

```
class Point
{
    int x, y;
public :
    Point(int abs=0, int ord =0){x=abs;y=ord;}
    virtual void afficher()
    {
        cout << "(" << x << ","  << y << ")" ;
    }
};

class Pixel : public Point
{
    char *couleur;
public :
    Pixel(int abs=0, int ord=0, char *color="INCOLORE")
        :Point(abs,ord),couleur(color){};
    void afficher()
    {
```

```
            cout << "[" ; Point::afficher() ; cout << ";" << couleur << "] \n";
        }
    } ;
```

Utilisation:

```
Pixel *pix=new Pixel(1,2,"BLEU");
Point *pt=new Point(10,20);

pt=pix ;
pt->afficher();                                    // pt « a » une couleur !!!
```

Les différentes redéfinitions d'une fonction peuvent être vues comme des versions de plus en plus spécialisées d'un traitement spécifié de manière générique dans la classe de base C.

Appeler une fonction virtuelle à travers un pointeur p sur un objet de C revient à demander l'exécution de la version la plus spécialisée de la fonction qui s'applique à ce que p pointe en fait à l'instant où l'appel est exécuté.

Contraintes de la redéfinition des fonctions virtuelles :

- la redéfinition d'une fonction virtuelle n'est pas obligatoire,
- la signature (liste des types des arguments, sans le type du résultat) de la redéfinition doit être strictement la même que celle de la première définition, sinon la deuxième définition n'est pas une redéfinition mais une surcharge de la première,

En Java, la ligature des fonctions est toujours dynamique. La notion de fonction virtuelle n'existe pas : tout se passe en fait comme si toutes les fonctions membres étaient virtuelles. De plus, comme toute classe est toujours dérivée de la class Object, deux classes différentes appartiennent toujours à une même hiérarchie. Le polymorphisme est donc toujours effectif en Java.

5.8.Classes abstraites

En POO, il est possible de définir des classes qui serviront, non pas à instancier des objets mais à engendrer d'autres classes par héritage, qui permettront d'instancier des objets.

Les fonctions virtuelles sont souvent introduites dans des classes placées à des niveaux si élevés de la hiérarchie d'héritage qu'on ne peut pas leur donner une définition.

Une première solution consiste à en faire des fonctions vides :

```
class Figure
{
…
```

```
public:
…
    virtual void identifier() { }          // fonction « vide »
                                           // qui sera définie dans les classes
                                           // dérivées
} ;
```

Deux problèmes apparaissent alors :

- un utilisateur peut déclarer un objet de classe Figure, alors que dans l'idée du programmeur, il s'agissait d'une classe abstraite, l'appel de identifier pour un tel objet conduira à un appel d'une fonction ne faisant rien,
- la fonction identifier pourrait ne pas être redéfinie dans les classes dérivées ce qui conduit au même problème.

Une manière de résoudre ces problèmes consiste à faire appel à la notion de fonction virtuelle pure et de classe abstraite.

Une fonction virtuelle pure est une fonction qui est déclarée nulle :

```
class Figure
{
…
public:
…
    virtual void identifier() = 0 ;        // fonction virtuelle pure
} ;
```

Typiquement, une fonction virtuelle pure n'a pas de définition, rien n'empêche cependant d'ajouter une implémentation dans la classe dans laquelle elle est déclarée.

C++ adopte les règles suivantes :

- une classe comportant au moins une fonction virtuelle pure est considérée comme abstraite et il n'est plus possible de déclarer des objets de son type,
- une fonction déclarée virtuelle pure dans une classe de base doit obligatoirement être redéfinie dans une classe dérivée ou déclarée à nouveau virtuelle pure, dans ce dernier cas, la classe dérivée est elle aussi abstraite.

Pour le compilateur, une *classe abstraite* est une classe qui a des fonctions virtuelles pures. Tenter de créer des objets d'une classe abstraite est une erreur, qu'il signale :

```
class Figure
{
…
public:
…
    virtual void identifier() = 0 ;        // Une fonction virtuelle pure
} ;
```

```
class Carre : public Figure
{
…
public :
…
      void identifier()
      {
            cout << "Je suis un carré "<< endl ;
      }
} ;
```

Utilisation :

```
Figure f;
Figure *pt;
pt = new Figure;                    // Erreur : Création d'une
                                    // instance de la classe abstraite
                                    //Figure
Carre r;
pt = new Carre;                     // OK Carre n’est pas abstraite
```

En java, on peut définir explicitement une classe abstraite ou une méthode abstraite, à l’aide du mot clé abstract. En java, les méthodes abstraites n'ont pas d'implémentation.

6.Modèles (généricité ou templates)

Un modèle définit une famille de fonctions (fonctions génériques, patron de fonctions, fonctions templates) ou de classes paramétrées par une liste d'identificateurs qui représentent des valeurs et des types.

Les valeurs sont indiquées par leurs types respectifs, les types par le mot réservé class.

Le mot réservé template se place devant :

Exemple :

```
template<class T, class Q, int N>.
```

ici, T et Q apparaissent comme types et N comme constante.

T et Q sont donc des *paramètres de type*.

Le mot class ne signifie pas que T et Q sont nécessairement des classes, mais des types.

On peut aussi utiliser le mot réservé typename à la place du mot class :

```
template<typename T, typename Q, int N>.
```

La production effective (génération du code) d'un élément de la famille définie par un modèle s'appelle l'instanciation du modèle.

Lors de cette opération, les paramètres qui représentent des types doivent recevoir pour valeur des types; les autres paramètres doivent recevoir des expressions dont la valeur est connue durant la compilation.

6.1.Modèles de fonctions

Un modèle de fonction définit une famille de fonctions à partir d'une seule définition. Toutes les fonctions de la famille réalisent donc le même algorithme.

L'instanciation d'un modèle de fonction

template<$param_1$, ..., $param_n$>

type nom($argf_1$, ... $argf_n$)...

est commandée implicitement par l'appel d'une des fonctions que le modèle définit. S'il n'y a pas d'appel, il n'y a pas d'instanciation !

Exemple : déclaration du modèle recherche de la valeur minimale d'un tableau (ayant au moins un élément) :

```
template<class T> T min(T tab[ ], int n)
{
    T min = tab[0];
    for (int i = 1; i < n; i++)
    if (tab[i] < min) min = tab[i];
    return min;
}
```

Voici un programme qui produit et utilise deux instances de ce modèle :

```
int t[ ] = {15, 8, 2, 3, 51, 4, 7};
cout << min<int>(t, 7);                          // instanciation : T = int
cout << min<char>("CVETASUPLKTG", 12);           // instanciation : T = char
```

Puisque dans les deux cas l'argument du modèle peut se déduire de l'appel, ce programme s'écrit aussi :

```
cout  << min(t, 7);                              // T = int
cout << min("CVETASUPLKTG", 12);                 // T =  char
```

Un modèle de fonction peut coexister avec une ou plusieurs spécialisations.

Par exemple, la fonction suivante est une spécialisation du modèle min qu'on ne peut pas obtenir par instanciation de ce modèle, car à la place de l'opérateur de comparaison < figure un appel de la fonction strcmp :

```
char *min(char *tab[ ], int n)
{
    char *min = tab[0];
    for (int i = 1; i < n; i++)
    if (strcmp(tab[i], min) < 0)
        min = tab[i];
    return min;
}
```

Suite au mécanisme de surcharge des fonctions, les instances de modèles sont souvent en concurrence avec d'autres instances ou des fonctions ordinaires.

Les fonctions ordinaires sont préférées aux instances de modèles et les instances de modèles plus spécialisés sont préférées aux instances de modèles moins spécialisés.

Exemple : écrivons une fonction qui recherche l'indice du minimum d'un tableau, avec un deuxième tableau pour critère secondaire.

Nous définissons :

1. Un modèle avec deux paramètres-types :

```
template<class T1, class T2> int ind_min(T1 ta[ ], T2 tb[ ], int n)
{
```

```
    cout << "ind_min(T1[ ], T2[ ], int) : ";
    int m = 0;
    for (int i = 1; i < n; i++)
        if (ta[i] < ta[m] || ta[i] == ta[m] && tb[i] < tb[m])
            m = i;
    return m;
}
```

2. Un modèle qui en est une spécialisation partielle (le premier tableau est un tableau de chaînes, il n'y a donc plus qu'un paramètre de type) :

```
template<class T> int ind_min(char *ta[ ], T tb[ ], int n)
{
    cout << "ind_min(char *[ ], T[ ], int) : ";
    int m = 0, r;
    for (int i = 1; i < n; i++)
    {
        r = strcmp(ta[i], ta [m]);
        if (r < 0 || r == 0 && tb [i] < tb [m] )
            m =  i;
    }
    return m;
}
```

3. Une fonction qui est une spécialisation complète du modèle (deux tableaux de chaînes, plus aucun paramètre-type) :

```
int ind_min(char *ta[ ], char *tb [ ], int n)
{
    cout << "ind_min(char *[ ], char *[ ], int) : ";
    int m = 0, r;
    for (int i = 1; i < n; i++)
    {
        r = strcmp(ta[i], ta[m]);
        if (r < 0 || r == 0 && strcmp(tb[i], tb[m]) < 0)
            m =  i;
    }
    return m;
    }
```

Utilisation :

```
int ti1[5] = { 10, 5, 15, 5, 18 };

int ti2[5] = {20, 8, 20, 10, 20 };

char *ts1[5] = {"Andre", "Denis", "Charles", "Andre", "Bernard"};

char *ts2[5]   = {"Duchamp", "Dubois", "Dumont", "Dupont", "Durand"  };

cout << ind_min(ti1, ti2, 5) << endl;

cout << ind_min(ts1, ti2, 5) << endl;

cout << ind_min(ts1, ts2, 5) << endl;
```

Affichage obtenu :

ind_min(T1[ ], T2[ ], int) : 1

ind_min(char *[ ], T[ ], int) : 3

ind_min(char *[ ], char *[ ], int) : 0

Lors de l'appel d'une fonction, il doit y avoir correspondance absolue des types.

Exemple :

```
template<typename T> T max(Ta,Tb) {return a>b ? a : b ;}
...
int i=3, j=4 ;
char a='a', b='b' ;
max(i,j) ;                                  // OK
max(a,b)                                    // OK
max(i,a)                                    // Erreur
```

Il est indispensable que le paramètre de type apparaisse au moins une fois parmi les types des arguments de la fonction.

Exemple :

```
template<typename T> T fonction(int, char)...      // Erreur : ambiguïté d'instanciation
```

6.2.Modèles de classes

Un modèle de classe est un type paramétré, par d'autres types et par des valeurs connues lors de la compilation.

Exemple :

```
template <typename T> class Point                  //ou template <class T> classPoint
{
      T x;
      T y;
public:
      Point(T abs=0, T ord=0) ;
      void afficher() ;
      ...;

} ;
```

Utilisation :

```
Point <int> p1(1,2) ;
Point <float> p2(2.5, 3.8) ;
```

La première déclaration instancie la définition d'une classe Point dans laquelle le

paramètre T prend la valeur int. Dans la seconde déclaration, l'instanciation a lieu avec float comme valeur du paramètre T.

Les paramètres des modèles de classes peuvent avoir des valeurs par défaut comme le montre l'exemple ci-dessus.

6.2.1.Fonctions membres non génériques d'une classe générique

Les fonctions membres d'un modèle de classe sont des modèles de fonctions avec les mêmes paramètres que le modèle de la classe.

Cela est implicite pour les déclarations et définitions écrites à l'intérieur du modèle de classe.

Exemple :

```
template <typename T> class Point
{
      T x;
      T y;
public:
      Point(T abs=0, T ord=0) ;
      void afficher()
      {
            cout << "(" << x << ","  << y << ")" ;
      }

      ...;

} ;
```

Lorsque la définition de la fonction membre se trouve en dehors de la classe, il est nécessaire de :

- fournir les paramètres de type sous la forme template <typename T>,
- le nom du modèle concerné : Point <T> .

Exemple :

```
      template <typename T> void Point<T> ::afficher()
      {
            cout << "(" << x << ","  << y << ")" ;
      }
```

6.3.Fonctions membres génériques (template)

A l'exception des destructeurs, les fonctions membres d'une classe peuvent être *template*.

Les fonctions membres virtuelles ne peuvent pas être *template*. Si une fonction membre *template* a le même nom qu'une fonction membre virtuelle d'une classe de base, elle ne constitue pas une redéfinition de cette fonction.

Les fonctions membres *template* peuvent appartenir à une classe *template* ou à une classe normale.

6.3.1.Fonctions membres génériques d'une classe non générique

Lorsque la classe à laquelle elles appartiennent n'est pas *template*, leur syntaxe est similaire à celle utilisée pour les fonctions *template* non membre.

Exemple :

```
class A
{
      int i;
public :
template <class T>
void ajoute(T valeur);
};



template <class T>
void A::ajoute(T valeur)
{
      i=i+((int) valeur);
return ;
}
```

6.3.2.Fonctions membres génériques d'une classe générique

Si la classe dont la fonction membre fait partie est elle aussi *template*, il faut répéter deux fois la déclaration *template* : une fois pour la classe, et une fois pour la fonction.

Si la fonction membre *template* est définie à l'intérieur de la classe, il n'est pas nécessaire de donner les paramètres *template* de la classe, et la définition de la fonction membre *template* se fait donc exactement comme celle d'une fonction *template* classique.

Exemple :

```
template<class T>
class string
{
public:
```

```
/* Fonction membre template définie à l'extérieur de la classe template : */

    template<class T2> int compare(const T2 &);

/* Fonction membre template définie à l'intérieur de la classe template : */

    template<class T2>
    string(const string<T2> &s)
    {
    ...
    }
};
```

À l'extérieur de la classe template, il est nécessaire de donner les déclarations template pour la classe et pour la fonction membre template :

```
template<class T> template<class T2>
int string<T>::compare(const T2 &s)
{
    ...
}
```

Remarque :

Les fonctions membres virtuelles ne peuvent pas être *template*. Il est néanmoins possible de définir une fonction membre virtuelle non *template* qui appellera la fonction membre *template*.

6.3.3.Organisation du code source

Habituellement, la déclaration d'une classe se fait dans un fichier et la définition des méthodes déportées dans un autre.

Dans le cas de fonctions ou de classes template, il est nécessaire d’écrire le code des méthodes à la suite de la déclaration de la fonction ou de la classe.

Le code template n'est qu'un modèle, lorsque le template est instancié, le compilateur doit avoir accès à l'intégralité du code afin de générer la version qui correspond au type passé en paramètre.

7.Exceptions

7.1.Principe et syntaxe

Même lorsqu'il est bien conçu, un programme peut rencontrer des conditions exceptionnelles qui risquent de compromettre la poursuite de son exécution.

Dans les programmes conséquents où la réutilisabilité joue une place importante, il devient essentiel de séparer la détection d'un incident et son traitement.

Une exception est une rupture de séquence déclenchée par une instruction throw.

Il y a alors branchement à un ensemble d'instructions nommé gestionnaire d'exceptions, dont le nom est déterminé par la nature de l'exception.

Les points-clés de ce mécanisme sont les suivants :

- la fonction qui détecte un événement exceptionnel construit une exception et la lance *(throw)* vers la fonction qui l'a appelée ;
- l'exception est un nombre, une chaîne, idéalement un objet d'une classe spécialement définie dans ce but (souvent une classe dérivée de la classe exception) comportant diverses informations utiles à la caractérisation de l'événement à signaler ;
- une fois lancée, l'exception traverse la fonction qui l'a lancée, sa fonction appelante, la fonction appelante de la fonction appelante, etc., jusqu'à atteindre une fonction active qui a prévu d'attraper *(catch)* ce type d'exception;
- lors du lancement d'une exception, la fonction qui l'a lancée et les fonctions que l'exception traverse sont immédiatement terminées : les instructions qui restaient à exécuter dans chacune de ces fonctions sont abandonnées; cette terminaison prend le temps de détruire les objets locaux de chacune des fonctions ainsi avortées ;
- si une exception arrive à traverser toutes les fonctions actives, car aucune de ces fonctions n'a prévu de l'attraper, alors elle produit la terminaison du programme.

Ce mécanisme se traduit par un bloc try suivi d'un ensemble de gestionnaires d'exceptions ou gestionnaires catch :

```
try {
    instructions susceptibles de provoquer, soit directement soit dans des
    fonctions appelées, le lancement d'une exception
}
catch(type1 Paramètre1)  {
    instructions pour traiter les exceptions correspondant au type de
    Paramètre1
}
```

catch(type2 Paramètre2) {

instructions pour traiter les exceptions, non attrapées par le gestionnaire précédent, correspondant au type de Paramètre2

}

...

catch(...) {

instructions pour traiter toutes les exceptions non attrapées par les gestionnaires précédents

}

Un bloc try doit être immédiatement suivi par au moins un gestionnaire catch.

Un gestionnaire catch doit se trouver immédiatement après un bloc try ou immédiatement après un autre gestionnaire catch.

Chaque gestionnaire catch comporte un en-tête avec la déclaration d'un argument formel qui sera initialisé, à la manière d'un argument d'une fonction, avec l'exception ayant activé le gestionnaire.

Exemple :

```
catch(bad_alloc e)
{
    cerr << "Erreur : " << e.whatQ << endl;
}
```

Si le corps du gestionnaire ne référence pas l'exception en question, l'en-tête peut se limiter à spécifier le type de l'exception :

```
catch(bad_alloc)
{
    cout << "Erreur : Allocation impossible";
}
```

Comme il a déjà dit, l'exception est propagée aux fonctions actives, jusqu'à trouver un gestionnaire catch dont le paramètre indique un type compatible avec celui de l'expression. Les fonctions sont explorées de la plus récemment appelée vers la plus anciennement appelée (c.-à-d. du sommet vers la base de la pile d'exécution) ; à l'intérieur d'une fonction, les gestionnaires catch sont explorés dans l'ordre où ils sont écrits. Le gestionnaire catch(...) attrape *toutes* les exceptions. Il n'est donc pas toujours présent et, quand il l'est, il est le dernier de son groupe.

Dès l'entrée dans un gestionnaire catch, l'exception est considérée traitée.

A la fin de l'exécution d'un gestionnaire, le contrôle est passé à l'instruction qui suit le dernier gestionnaire de son groupe.

Les instructions restant à exécuter dans la fonction qui a lancé l'exception et dans les fonctions entre celle-là et celle qui contient le gestionnaire qui a attrapé l'exception sont abandonnées.

Les objets locaux attachés aux blocs brusquement abandonnés à cause du lancement d'une exception sont proprement détruits.

7.2.Attraper une exception

Lorsqu'une exception est lancée, un gestionnaire catch ayant un argument compatible avec l'exception est recherché dans les fonctions actives (commencées et non encore terminées), de la plus récemment appelée vers la plus anciennement appelée.

Un argument d'un gestionnaire catch de type T1, const T1, T1& ou const T1& est compatible avec une exception de type T2 si et seulement si :

- T1 et T2 sont le même type, ou
- T1 est une classe de base accessible de T2, ou
- T1 et T2 sont de la forme B* et D* respectivement, et B est une classe de base accessible de D.

Les conversions standard sur des types qui ne sont pas des pointeurs ne sont pas effectuées. Ainsi, par exemple, un gestionnaire catch(double) n'attrape pas des exceptions int.

7.3.La classe exception

Une exception est une valeur quelconque, d'un type prédéfini ou bien d'un type classe défini à cet effet.

Les exceptions lancées par les fonctions de la bibliothèque standard sont toutes des objets de classes dérivées de la classe exception. Elles sont déclarées dans le fichier en-tête <stdexcept>.

La classe exception comporte au minimum les membres suivants :

```
class exception
{
public:
      exception() throw();
      exception(const exception &e) throw();
      exception &operator=(const exception &e) throw();
      virtual ~exception() throw() ;
      virtual const char *what() const throw();
};
```

Les quatre premiers membres correspondent aux besoins internes de l'implémentation des exceptions.

La fonction what renvoie une chaîne de caractères qui est une description informelle de l'exception.

Les classes qui dérivent de la classe exception sont organisées suivant la hiérarchie ci-dessous :

exception :

- bad_alloc
- bad_cast
- bad_exception
- bad_typeid
- logic_error:
  - domain_error
  - invalid_argument
  - length_error
  - out_of_range
- runtime_error:
  - range_error
  - overflow_error
  - underflow_error

Les exceptions susceptibles d'être lancées par une fonction ou un opérateur de la bibliothèque standard sont les suivantes:

- bad_alloc : échec d'une allocation de mémoire par new,
- bad_cast : échec de l'opérateur dynamic_cast,
- bad_typeid : échec de la fonction typeid,
- badexception : signale l'émission par une fonction d'une exception qui n'est pas déclarée dans sa clause throw.
- out_of_range : erreur d'indice,
- invalid_argument : erreur d'argument,
- rangeerror : erreur de rang,
- overflow_error : débordement arithmétique (par le haut).

8.Flots

8.1.Présentation générale

D'une manière générale, un flot peut être considéré comme un « canal » :

- recevant de l'information – flot de sortie,
- fournissant de l'information – flot d'entrée.

Les opérateurs << ou >> servent à assurer le transfert de l'information.

Un flot peut être connecté à un périphérique ou à un fichier.

Le flot prédéfini cout est connecté à la sortie standard.

Le flot prédéfini cin est connecté à l'entrée standard

Généralement, l'entrée standard correspond par défaut au clavier et la sortie standard à l'écran.

Un flot d'entrées-sorties est un objet d'une classe prédéfinie:

- ostream pour un flot de sortie,
- istream pour un flot d'entrée.

Chacune de ces deux classes surcharge les opérateurs << et >> pour les différents types de base.

Leur emploi nécessite l'inclusion du fichier en-tête iostream.

8.2.Classes de flots

Les classes de flots sont au nombre de dix-huit, réparties dans trois fichiers distincts : <iostream>, <fstream> et <strstream>.

Un quatrième fichier <iomanip> peut être utilisé dans certains cas.

On peut distinguer les catégories suivantes de flots :

- Les *tampons* d'entrées-sorties, divisés en trois classes *streambuf*, *strstreambuf* et *filebuf,*
- Les *flots* d'entrées-sorties.

Un flot d'entrées-sorties est une liste de caractères qu'on ne charge pas entièrement en mémoire.

Un flot doit donc avoir un *tampon*, c'est-à-dire un petit bloc de mémoire qui permet de ranger les caractères en attente.

Ce tampon est géré par des opérations de bas niveau d'un élément de la classe streambuf ou de ses dérivées.

Chaque flot contient un pointeur sur un tampon (ou plusieurs éventuellement), plus un certain nombre de renseignements supplémentaires qui indiquent notamment l'état dans lequel il se trouve.

Le type de tampon détermine en grande partie le type de flot tandis que le type de flot indique les opérations autorisées.

8.3.Flots généraux : classe ios

La classe ios est la base des flots, elle ne permet qu'un petit nombre d'opérations. Ce n'est généralement pas la classe de base ios qui est instanciée mais plutôt les classes dérivées iostream et istream ou d'autres classes dérivées.

Une instance de ios est associée à un tampon streambuf.

La fonction membre streambuf* rdbuf() renvoie un pointeur sur ce tampon.

D'autres fonctions membres (et leurs redéfinitions) et données membres de la classe ios sont très largement utilisées dans les classes dérivées.

8.3.1.État des flots

Une énumération dans ios contient une liste de bits qui indiquent l'état du flot.

Bits indicateurs:

ios::goodbit	Lorsque ce bit vaut 0, ainsi que tous les autres, tout va bien.
ios::eofbit	Lorsque ce bit vaut 1, la fin du fichier est atteinte
ios::failbit	Ce bit est à 1 lorsqu'une opération a échoué. Le flot peut être réutilisé.
ios::badbit	Ce bit est à 1 lorsqu'une opération invalide a été tentée ; en principe le flot peut continuer à être utilisé mais ce n'est pas certain.
ios::hardfail	Ce bit est à 1 lorsqu'une erreur grave s'est produite ; il ne faut plus utiliser le flot.

Fonctions membres :

- int good(void) renvoie 1 si tous les bits d'état sont à zéro (tout va bien), 0 sinon.
- int eof() renvoie 1 dans ce cas, 0 sinon.
- int bad(void) renvoie 1 si l'un des deux bits ios::badbit ou ios::hardfail est à 1, 0 sinon.
- int fail(void) renvoie 1 si l'un des trois bits ios::badbit ou ios::failbit ou ios::hardfail est à 1, et 0 sinon.
- void clear(int i = 0) permet de modifier l'état du flot. Par exemple, l'écriture fl.clear(ios::failbit) positionne le bit ios::failbit du flot fl, indiquant une erreur grave.

8.3.2.Mode d'écriture

D'autres bits indiquent de quelle façon les données sont lues ou écrites, et ce qui se passe au moment de l'ouverture du flot.

Voici la liste de ces bits :

ios::in	Fichier ouvert en lecture.
ios::out	Fichier ouvert en écriture.
ios::app	Ajoute les données, en écrivant toujours à la fin (et non à la position courante).
ios::ate	Aller à la fin du fichier à l'ouverture (au lieu de rester au début).
ios::trunc	Supprime le contenu du fichier, s'il existe déjà ; cette suppression est automatique pour les fichiers ouverts en écriture, sauf si ios::ate ou ios::app a été précisé dans le mode.
ios::nocreate	Pour une ouverture en écriture, ne crée pas le fichier s'il n'existe pas déjà ; une erreur (bit ios::failbit positionné) est produite dans le cas où le fichier n'existe pas encore.
ios::noreplace	Pour une ouverture en écriture, si ni ios::ate ni ios::app ne sont positionnés, le fichier n'est pas ouvert s'il existe déjà, et une erreur est produite.
ios::binary	Fichier binaire, ne faire aucun formatage.

Exemple:

```
fstream fl("fich.cpp", ios::in|ios::out|ios::app);
```

ouvre le fichier fich.cpp en lecture et écriture, avec ajout des nouvelles données à la fin du fichier.

8.3.3.Indicateurs de format

Les flots permettent un grand nombre de formatages des données.

ios::skipws	Supprime les espaces (blancs, tabulations, etc.) en lecture. Ce bit est à 1 par défaut, contrairement aux autres.
ios::left	Ajustement à gauche en écriture.
ios::right	Ajustement à droite en écriture.
ios::internal	Remplissage après le signe + ou -, ou l'indicateur de base (et non avant).
ios::dec	Écriture décimale (base 10).
ios::oct	Écriture en octal (base 8).
ios::hex	Écriture en hexadécimal (base 16).
ios::showbase	En écriture, écrire un indicateur de base.
ios::showpoint	Écrit obligatoirement le point décimal pour les nombres à virgule flottante, même si toutes les décimales sont nulles.
ios::uppercase	Écrit les lettres A à F en majuscules dans les chiffres hexadécimaux (minuscules sinon).
ios::showpos	Écrit le signe pour tous les entiers, même positifs.
ios::scientific	Pour les nombres à virgule flottante, écriture en notation scientifique (1.75^{e}+01 par exemple).
ios::fixed	Pour les nombres à virgule flottante, écriture en notation à virgule flottante (17.5 avec le même exemple).
ios::unitbuf	Vide les tampons après écriture.

ios::stdio	Vide les tampons de sortie standard out et de sortie d'erreur err après insertion.

Le champ de format peut être lu par la méthode long flags(void) et modifié par long flags(long).

Deux autres méthodes peuvent être utilisées à cette fin. La méthode long unsetf(long) met à zéro les bits du champ de format qui valent 1 dans son argument. La méthode long setf(long) a l'effet contraire.

Mais on utilisera surtout long setf(long, long) ; le premier argument indique la nouvelle valeur des bits à modifier ; le deuxième argument indique les bits à modifier effectivement (ceux à 1 ; ceux à 0 ne sont pas changés), et peut être pris dans la liste de constantes suivante :

ios::basefield	égal à	ios::dec | ios::oct | ios::hex
ios::adjustfield	égal à	ios::left|ios::right|ios::internal
ios::floatfield	égal à	ios::scientific | ios::fixed

Par exemple

```
f.setf(ios::uppercase|ios::hex, ios::uppercase|ios::basefield);
```

change les bits de base d'écriture et de uppercase afin d'écrire les entiers en hexadécimal avec des lettres majuscules pour les chiffres A à F.

Trois autres champs de formatage peuvent être utilisés.

Le *champ de largeur* indique sur combien de caractères de large la donnée doit être écrite ; si la donnée est plus petite, la partie restante est remplie avec le caractère de remplissage ; si elle est trop grande elle *n'est pas* tronquée. Ce champ est remis à zéro après chaque opération formatée. Il peut être lu par la méthode int width(void) et modifié par int width(int) (qui renvoie la valeur précédente).

Le *champ de remplissage* indique quel caractère est utilisé pour le remplissage lorsqu'il y en a un. Par défaut, c'est l'espace blanc qui est utilisé. Ce champ peut être lu par char fill(void) et modifié par char fill(char).

Le *champ de précision* indique combien de décimales sont écrites au maximum dans les nombres à virgule flottante. Par défaut, le plus grand nombre de décimales significatives est écrit. Ce champ peut être lu par int precision(void) et modifié par int precision(int).

Voici quelques exemples :

```
int i = 245;
double d = 75.8901;
```

```
cout.precision(2);
cout.setf(ios::scientific, ios::floatfield);
cout << d;                                          // écrit 7.59e+01
cout.width(7);
cout.fill('$');
cout << i;                                          // écrit $$$$245
cout.width(9);
cout.fill('#');
cout.setf(ios::left|ios::hex,ios::adjustfield|ios::basefield);
cout << i;                                          // écrit f5#######
cout.fill(' ');
cout.width(6);
cout.setf(ios::internal|ios::showpos|ios::dec,ios::adjustfield
 |              ios::showpos|ios::basefield);
cout << i;                                          // écrit + 245
```

8.4.Manipulateurs

Il est possible de formater de différentes façons les entrées-sorties à l'aide du champ de format, du champ de largeur et du champ de remplissage.

Cependant, les écritures deviennent vite assez lourdes.

Pour les simplifier, on dispose de *manipulateurs*.

Ceux qui sont définis dans <iostream> sont les suivants :

endl	(sorties) Passe à la ligne et vide le tampon.
ends	(sorties) Insère un caractère nul.
flush	(sorties) Vide le tampon.
dec	Mode décimal.
hex	Mode hexadécimal.
oct	Mode octal.
ws	(entrées) Supprime les espaces.

Pour les employer, il suffit de les écrire sur le flot de la même façon qu'un objet normal, au moment où l'on souhaite changer le mode.

Par exemple, l'écriture suivante :

```
cout.setf(ios::dec, ios::basefield);
cout << i;
cout.setf(ios::hex, ios::basefield);
cout << j << '\n';
```

sera simplifiée ainsi :

```
cout << dec << i << hex << j << endl;
```

avec le même effet.

Le fichier <iomanip> contient des manipulateurs supplémentaires prenant des paramètres :

setbase(int)	Fixe la base d'écriture ou de lecture ; les valeurs admises sont 8 (octal), 10 (décimal), 16 (hexadécimal), et 0 qui indique un comportement standard : sorties en décimal sauf pour les pointeurs, entrées suivant le préfixe.
setfill(char)	Fixe le caractère de remplissage.
setprecision(int)	Fixe la précision (nombre de décimales en virgule flottante).
setw(int)	Fixe le champ de largeur width.
resetiosflags(long)	Met à zéro dans le champ de forme les bits qui sont à 1 dans le paramètre.
setiosflags(long)	Met à 1 dans le champ de forme les bits qui sont à 1 dans le paramètre.

On pourra donc écrire par exemple :

```
int i = 32;
cout << setfill('*') << setw(9) << hex << i;          // écrit : *******20
double d = 1.2303;
cout << setprecision(3) << d;                         // écrit : 1.230
```

9.La bibliothèque standard

9.1.Conteneurs et itérateurs

La bibliothèque standard fournit un ensemble de classes appelées conteneurs.

Les conteneurs permettent de représenter les structures de données les plus répandues telles que les vecteurs, les listes, les ensembles ou les tableaux associatifs.

Il s'agit de patrons de classes paramétrés par le type de leurs éléments.

Par exemple, on pourra construire une liste d'entiers, un vecteur de flottants ou une liste de points (point étant une classe) par les déclarations suivantes :

```
list <int> li ;                    //liste vide d'éléments de type int.
vector <double> ld ;               //vecteur vide d'éléments de type
                                   //double.
list <point> lp ;                  //liste vide d'éléments de type
                                   //point.
```

La norme C++ classe les différents conteneurs en deux catégories :

- les conteneurs séquentiels,
- les conteneurs associatifs.

Un itérateur est un objet défini généralement par la classe conteneur concernée qui généralise la notion de pointeur :

- un itérateur possède une valeur qui désigne un élément donné d'un conteneur, l'itérateur pointe sur un élément d'un conteneur,
- un itérateur peut être incrémenté par l'opérateur ++,
- un itérateur peut être déréférencé, comme un pointeur, en utilisant l'opérateur *,
- deux itérateurs sur un même conteneur peuvent être comparés.

Tous les conteneurs fournissent un itérateur nommé iterator et possédant au minimum les propriétés ci-dessus qui correspondent à ce qu'on appelle itérateur unidirectionnel.

9.2.Conteneurs séquentiels

Les conteneurs séquentiels principaux sont les classes string (cf chapitre suivant) vector, list et deque:

- vector: tableau dont la taille varie, accès direct rapide aux éléments, insertion et suppression efficaces en fin de vecteur, éléments stockés de façon contigüe.
- list: accès direct aux éléments impossible, insertion et suppression d'un élément

quelle que soie sa position. Éléments stockés de façon non contigüe.

- deque: queue à double entrée, insertion d'éléments en début et fin du conteneur rapides. Les éléments ne sont pas stockés de manière contigüe.

Nous allons étudier les fonctionnalités communes de ces trois conteneurs qui concernent leur :

- construction,
- affectation globale,
- comparaison,
- insertion/suppression.

9.2.1.Constructeurs

Exemples :

```
list<fIoat> lf(5) ;                  //liste de 5 éléments de type float.
vector<point> vp (5) ;               //5 éléments de type point
                                     //initialisés par le constructeur sans
                                     //argument.
list<int> li(5, 999) ;               //éléments de type int ayant
                                     //la valeur 999.

Point a (3, 8) ;
list<point> lp (10, a) ;             //10 points ayant tous la valeur de
                                     //a : il y a appel du constructeur par
                                     //copie de Point */
```

9.2.2.Modifications globales

Les classes vector, list et deque permettent d'affecter un conteneur d'un type donné à un conteneur de même type.

Exemples :

```
vector<int> vi1 (...), vi2 (...) ;
vector<float> vf (...) ;
vector<point> vp1 (....), vp2 (...) ;

vi1 = vi2;                           //correct, quels que soient
                                     //le nombre d'éléments de vi1
                                     //et de vi2 ; le contenu de vi1 est
                                     //remplacé par celui de vi2.
vf = vi1 ;                           //incorrect.
```

```
vp1 = vp2 ;                                    //correct
```

9.2.3.Comparaison

Les conteneurs vector, list et deque définissent des opérateurs ==, < , !=, <=, > et >=. La définition de == correspond à ce qu'on attend d'un tel opérateur, tandis que celle de < se base sur une comparaison lexicographique.

Exemples :

```
int t1 [ ] = {2, 5, 2, 4, 8 };
int t2[ ] = {2, 5, 2, 8 } ;
vector<int> v1 (t1, t1+5) ;                    //v1 contient : 2 5 2 4 8.
vector<int> v2 (t2, t2+4) ;                    //v2 contient : 2 5 2 8.
vector<int> v3 (t2, t2+3) ;                    //v3 contient : 2 5 2.
vector<int> v4 (v3) ;                          //v4 contient : 2 5 2.
vector<int> v5 ;                               //v5 est vide.

v2 < v1 ; /* faux */        v3 < v2 ; / * vrai */        v3<v4 ; /* faux */
v4 < v3 ; /* faux */        v3 = v4 ; /* vrai */        v4 > v5 ; /* vrai */
v5 > v5 ; /* faux /         v5 < v5 ; /* faux */
```

9.2.4.Insertion/suppression

La fonction insert permet d'insérer :

- une valeur avant une position donnée :

  insert (position, valeur) : insère valeur avant l'élément pointé par position, fournit un itérateur sur l'élément inséré

- n fois une valeur donnée, avant une position donnée :

  insert (position, nb_fois, valeur) : insère nb_fois valeur, avant l'élément pointé par position fournit un itérateur sur l'élément inséré.

- les éléments d'un intervalle, à partir d'une position donnée :

  insert (debut, fin, position) : insère les valeurs de l'intervalle [debut, fin] avant l'élément pointé par position

La fonction erase permet de supprimer :

- un élément de position donnée :

  erase (position) : supprime l'élément désigné par position, fournit un itérateur sur l'élément suivant ou sur la fin de la séquence.

- les éléments d'un intervalle :

  erase (début, fin) : supprime les valeurs de l'intervalle [début, fin] , fournit un itérateur sur l'élément suivant ou sur la fin de la séquence.

9.3.Conteneurs associatifs

Les conteneurs associatifs ont pour principale vocation de retrouver une information, non plus en fonction de sa place dans le conteneur, mais en fonction de sa valeur ou d'une partie de sa valeur nommée clé.

Les deux conteneurs associatifs les plus importants sont map et multimap.

Tous deux associent une clé et une valeur : map impose l'unicité des clés, multimap ne l'impose pas, on pourra retrouver deux éléments ayant la même clé et qui apparaîtront alors consécutivement.

9.3.1.Le conteneur map

Le conteneur map est formé d'éléments composés de deux parties : une clé et une valeur.

Les éléments de ce container sont rangés dans l'ordre croissant des clés.

Exemple :

```
map<char, int> m ;          //conteneur de type map :
                            //clés de type char, valeurs
                            //de type int.
m['T'] = 6 ;                //insertion d'un élément formé
                            //par l'association de la clé T
                            //et de la valeur 6.
cout << m['P'] ;            //création de m['P'], initialisation
                            //de la valeur associée à 0.
```

9.3.2.Le conteneur multimap

Les possibilités des conteneurs map se généralisent aux conteneurs multimap qui possèdent les mêmes fonctions membres.

Un certain nombre de fonctions membres de la classe map, prennent tout leur intérêt lorsqu'on les applique à un conteneur multimap.

On peut, en effet :

- connaître le nombre d'éléments ayant une clé équivalente à une clé donnée, à l'aide de count (clé) ;
- obtenir des informations concernant l'intervalle d'éléments ayant une clé équivalente à une clé donnée, à savoir :

  lower_bound (clé) : fournit un itérateur sur le premier élément ayant une clé équivalente à clé.

  upper_bound (clé) : fournit un itérateur sur le dernier élément ayant une clé équivalente à clé.

  equal_range (clé) : fournit une paire formée des valeurs des deux itérateurs précédents, lower_bound (clé) et upper _bound (clé).

9.4.Algorithmes

Les algorithmes standard se présentent sous forme de patrons de fonctions. Leur code est écrit, sans connaissance précise des éléments qu'ils seront amenés à manipuler.

A titre indicatif, nous citerons quelques uns des algorithmes standard.

9.4.1.Algorithmes d'initialisation de séquences existantes

9.4.2.Algorithmes de recherche

9.4.3.Algorithmes de transformation

9.4.4.Algorithmes de tri

9.4.5.Algorithmes de recherche et fusion

10.La classe string

La bibliothèque standard C++ définit des types complémentaires sous forme de classes C++.

Le type string, permet de manipuler les chaînes de caractères de manière plus simple et plus sûre qu'avec des pointeurs et des tableaux de caractères.

Cette classe encapsule les chaînes de caractères C classiques elle dispose des caractéristiques suivantes :

- compatibilité quasi-totale avec les chaînes de caractères C standards,
- gestion des chaînes à taille variable,
- prise en charge de l'allocation dynamique de la mémoire et de sa libération en fonction des besoins et de la taille des chaînes manipulées,
- définition des opérateurs de concaténation, de comparaison et des principales méthodes de recherche dans les chaînes de caractères,
- intégration totale dans la bibliothèque standard, en particulier au niveau des flux d'entrée / sortie.

10.1.Construction et initialisation d'une chaîne

La manière la plus simple de construire une string est de la déclarer, sans paramètres, le constructeur par défaut est appelé, et la chaîne est initialisée à la chaîne vide.

Plusieurs autres possibilités existent pour initialiser une chaîne.

Le constructeur le plus courant est sans doute le constructeur qui prend en paramètre une chaîne de caractères C classique :

```
string chaine("Valeur initiale");
```

10.2. Accesseurs

La classe string fournit plusieurs accesseurs permettant d'obtenir des informations sur son état et sur la chaîne de caractères qu'elle contient.

Exemples :

```
string s("Bonjour"):
s.size(); s.length()          //renvoient la longueur de s
s.empty() ;                   //permet de savoir si s est vide
```

10.3.Modification de la taille des chaînes

Certaines fonctions membres permettent de modifier une chaîne pour la réduire, l'agrandir ou augmenter sa capacité.

```
#include <string>
#include <string.h>                    // pour strlen
...
string s("abc");
s.resize(10);                          // ajout de 7 caractères nuls
s.length() ;                           // longueur de s égale 10
strlen(s.c_str()) ;                    // longueur de la chaîne C
                                       // contenue dans s égale à 3
```

La méthode c_str permet d'obtenir l'adresse de la chaîne C stockée en interne dans la string.

```
s.resize(5, 'd');                      // s contient abcddddd
```

10.4.Accès aux données de la chaîne de caractères

La classe string surcharge l'opérateur d'accès aux éléments d'un tableau :

```
string s("alphanumérique”);
s[1]=’e’                               // remplace le deuxième caractère
                                       //de la chaîne par un 'e'.
```

L’opérateur [] renvoie la référence du caractère dont l’indice est spécifié en paramètre.

10.5.Opérations sur les chaînes

L’intérêt de la classe string est qu’elle fournit un ensemble de méthodes permettant d'effectuer les opérations les plus courantes sur les chaînes de caractères.

10.5.1.Affectation et concaténation

Plusieurs surcharges de l'opérateur d'affectation sont définies dans la classe string.

```
string s1("Hello you"),s2,s3;
s2=s3=s1;
```

La méthode assign, permet de réaliser des affectations plus complexes.

Exemples :

```
string s1,s2;
char* c=”123456”;
```

```
s1.assign(c+1,c+2);                    // affecte 23 à s1
s2.assign(s1,0,2);                     // affecte les deux premiers
                                       // caractères de s1 à s2
s1.assign(10,'A')                      // réinitialise s1 avec 10 'A'
```

De manière similaire, l'opérateur d'affectation avec addition '+=' a été surchargé afin de permettre les concaténations de chaînes de caractères.

Exemples :

```
string s1("Bonjour"),s2("à tous");
s1+=s2;                                // Bonjour à tous
```

L'ensemble de méthodes append permettent d'ajouter une partie d'une chaîne de caractères ou un nombre déterminé de caractères.

```
s1.append(" et à toutes.azerty ",13) ;  // Bonjour à tous et à toutes.
```

10.5.2.Insertion et suppression de caractères dans une chaîne

La méthode insert possède tout un jeu de surcharges.

Ces surcharges prennent toutes un paramètre en première position qui indique l'endroit où l'insertion doit être faite, les autres paramètres permettent de spécifier ce qui doit être inséré dans cette chaîne.

Exemples:

```
string s = "abef";
s.insert(2, "cdze", 2);                // Insère un 'c' et un 'd'
string gh = "gh";
s.insert(6, gh);                       // Idem pour 'g' et 'h'
```

Il existe diverses surcharges de la méthode erase, dont le but est de supprimer des caractères dans une chaîne en décalant les caractères suivant les caractères supprimés pour remplir les positions ainsi libérées.

Exemples:

```
string s = "abcdeerrfgh";
s.erase(5,3);                          // Supprime la faute de frappe
s.clear();                             // Efface la chaîne complète
if (s.empty()) cout << "Vide !" << endl;
```

10.5.3.Remplacements de caractères d'une chaîne

Ces méthodes nommées replace sont tout à fait similaires dans le principe aux méthodes insert.

Néanmoins, contrairement à celles-ci, les méthodes replace prennent un paramètre supplémentaire pour spécifier la longueur ou le caractère de fin de la sous-chaîne à remplacer.

Ce paramètre doit être fourni juste après le premier paramètre, qui indique toujours le caractère de début de la sous-chaîne à remplacer.

Les autres paramètres des fonctions replace permettent de décrire la chaîne de remplacement, et fonctionnent exactement comme les paramètres correspondants des fonctions insert.

Exemples :

```
string s = "abcerfg";
s.replace(3, 2, "de");                    // Remplace le 'e' et le 'r'
                                          // par un 'd' et un 'e'
```

La méthode swap de la classe string permet d'intervertir le contenu de deux chaînes de caractères.

Cette méthode prend en paramètre une référence sur la deuxième chaîne de caractères, avec laquelle l'interversion doit être faite.

Exemples :

```
string s1 = "abcd";
string s2 = "1234";
s1.swap(s2);                              // Intervertit les deux
chaînes
```

10.6. Recherche dans les chaînes

Les fonctions de recherche sont toutes surchargées afin de permettre de spécifier la position à partir de laquelle la recherche doit commencer d'une part, et le motif de caractère à rechercher.

Le premier paramètre indique toujours quel est ce motif, que ce soit une autre string, une chaîne de caractères C classique ou un simple caractère.

Exemples :

```
string s = "Bonjour tout le monde !";
string::size_type pos = s.find("monde");      // Recherche le mot "monde"
pos = s.rfind("tout");                        // Recherche le mot "tout"
                                              // en commençant par la fin
// Décomposition de la chaîne en mots :
string::size_type debut = s.find_first_not_of(" \t\n");
while (debut != string::npos)
```

```
{
    pos = s.find_first_of(" \t\n", debut);        // Recherche la fin du mot suivant
    if (pos != string::npos)                      // Affiche le mot
        cout << s.substr(debut, pos - debut) << endl;
    else
        cout << s.substr(debut) << endl;
    debut = s.find_first_not_of(" \t\n", pos);
```

Annexe : Allocation dynamique de mémoire et listes chaînées

Les pointeurs sont surtout utilisés pour créer un nombre quelconque de variables, ou des variables de taille quelconque, en cours d'exécution du programme.

Lorsque les variables à créer ainsi que leurs tailles sont connues au moment de la compilation, ces variables sont créées automatiquement lors de leur définition.

Par exemple, une ligne comme :

```
float tab[5];
```

signale au compilateur qu'une variable *tab* de 5 flottants doit être créée. Le programme s'en chargera donc automatiquement lors de l'exécution.

Si nous envisageons la gestion d'une liste de personnes, il n'est pas possible de savoir à l'avance combien de personnes seront entrées, le compilateur ne peut donc pas faire la réservation de l'espace mémoire automatiquement. C'est au programmeur de le faire.

Cette réservation de mémoire (appelée encore *allocation*) doit être faite pendant l'exécution du programme.

La différence avec la déclaration de tableau précédente, c'est que le nombre de personnes et donc la quantité de mémoire à allouer, est variable.

Il faut donc faire ce qu'on appelle une *allocation dynamique de mémoire*.

Allocation dynamique de mémoire en C

En C, il existe deux principales fonctions permettant de demander de la mémoire au système d'exploitation et de la lui restituer.

Ces fonctions sont déclarées dans le fichier stdlib.h, leur emploi nécessite la directive :

```
#include <stdlib.h>.
```

Elles utilisent toutes les deux les pointeurs, parce qu'une variable allouée dynamiquement n'a pas d'identificateur étant donné qu'elle n'était a priori pas connue à la compilation, et n'a donc pas pu être déclarée.

Les pointeurs utilisés par ces fonctions C n'ont pas de type.

Leur syntaxe est la suivante :

```
malloc(taille)

free(pointeur)
```

malloc (abréviation de « Memory ALLOCation ») alloue de la mémoire. Elle attend comme paramètre la taille de la zone de mémoire à allouer et renvoie un pointeur non typé (void *).

free (pour « FREE memory ») libère la mémoire allouée. Elle attend comme paramètre le pointeur sur la zone à libérer et ne renvoie rien.

Lorsqu'on alloue une variable typée, il est nécessaire de réaliser un transtypage du pointeur renvoyé par malloc en pointeur de ce type de variable.

Exemple : liste chaînée de personnes en C

```
#include <stdio.h>                          /*utilisation de printf et de scanf. */
#include <stdlib.h>                         /*utilisation de malloc et de free. */
#include <string.h>                         /*utilisation strcpy, strlen, strcmp. */


/* Type de base d'un élément de liste de personne. */
typedef struct person
{
   char *name;                              /* Nom de la personne. */
   char *address;                           /* Adresse de la personne. */
   struct person *next;                     /* Pointeur sur l'élément suivant. */
} Person;
typedef Person *Liste;                      /* Type de liste de personnes. */
/*********************************************************************************************/
/*********************************************************************************************/


                    /* Fonctions de gestion des listes de personnes : */
/*********************************************************************************************/
/*********************************************************************************************/
                /* Fonction d'initialisation d'une liste de personne.*/
void init_list(Liste *lst)
 {
   *lst = NULL;
 }
/*********************************************************************************************/
                        /* Fonction d'ajout d'une personne*/
int add_person(Liste *lst, const char *name, const char *address)
{
                                            /* Création d'un nouvel élément : */
   Person *p = (Person *) malloc(sizeof(Person));
   if (p != NULL)
```

```
    {       /* Allocation de la mémoire pour le nom et l'adresse.*/

            p->name = (char *) malloc((strlen(name) + 1) * sizeof(char));
            p->address = (char *) malloc((strlen(address) + 1) * sizeof(char));
            if (p->name != NULL && p->address != NULL)
            {
                    strcpy(p->name, name);          /* Copie le nom et l'adresse : */
                    strcpy(p->address, address);
                    p->next = *lst;
                    *lst = p;
            }
            else
            {
                    free(p);
                    p = NULL;
            }
    }
    return (p != NULL);
}
/*****************************************************************************************/

                        /* Fonction de suppression d'une personne.

La structure de la liste est modifiée par la suppression de l'élément de cette
personne. Cela peut impliquer la modification du chaînage de l'élément
précédent, ou la modification de la tête de liste elle-même. */

int remove_person(Liste *lst, const char *name)

{
        /* Recherche la personne et son antécédant : */
    Person *prev = NULL;
    Person *p = *lst;
    while (p != NULL)
    {
                    /* On sort si l'élément courant
                    est la personne recherchée : */
        if (strcmp(p->name, name) == 0)
        break;
                                                /* on passe à l'élément suivant
                                                sinon : */
        prev = p;
        p = p->next;
    }
    if (p != NULL)
    {
                                                /* La personne a été trouvée,
                                                on la supprime de la liste : */
        if (prev == NULL)
```

```
        {
                *lst = p->next;                 /*la personne est en tête de liste,
                                                mise à jour du pointeur de tête
                                                de liste*/ :
        }
        else
        {
                prev->next = p->next;           /* on met à jour le lien de l'élément
                                                précédent et on détruit la personne*/

        }
        free(p->name);
        free(p->address);
        free(p);
    }
    return (p != NULL);
}
/*****************************************************************************************/
                                /* Fonction d'affichage. */

        void print_list(Liste const *lst)
        {
                Person const *p = *lst;
                int i = 1;
                while (p != NULL)
                {
                        printf("Personne %d : %s (%s)\n", i, p->name, p->address);
                        p = p->next;
                        ++i;
                }
        }
/*****************************************************************************************/
            /* Fonction de destruction et de libération de la mémoire. */

        void destroy_list(Liste *lst)
        {
                while (*lst != NULL)
                {
                        Person *p = *lst;
                        *lst = p->next;
                        free(p->name);
                        free(p->address);
                        free(p);
                }
                return ;
        }
/*****************************************************************************************/
```

```
int main(void)
{
    int op = 0;
    size_t s;
    char buffer[16];
    char name[256];
    char address[256]
    Liste p ;                                   /* création d'une liste de personnes*/
    init_list(&p);
                                                /* utilisation de la liste : */
    do
    {
        printf("Opération (0 = quitter, 1 = ajouter, 2 = supprimer) ?");
        fgets(buffer, 16, stdin);
        buffer[15] = 0;
        op = 3;
        sscanf(buffer, "%d", &op);
        switch (op)
        {
            case 0:
                break;
            case 1:
                printf("Nom : ");
                fgets(name, 256, stdin);    /*lit le nom. */
                name[255] = 0;              /*caractère nul terminal*/
                s = strlen(name);
                if (name[s - 1] == '\n') name[s - 1] = 0;
                                            /* supprime l'éventuel saut de ligne
*/
                printf("Adresse : ");
                fgets(address, 256, stdin);
                address[255] = 0;
                s = strlen(address);
                if (address[s - 1] == '\n') address[s - 1] = 0;
                add_person(&p, name, address);
                break;
            case 2:
                printf("Nom : ");
                fgets(name, 256, stdin);
                name[255] = 0;
                s = strlen(name);
                if (name[s - 1] == '\n') name[s - 1] = 0;
                if (remove_person(&p, name) == 0)
                {
                    printf("Personne inconnue.\n");
                 }
```

```
                    break;
                default:
                    printf("Opération invalide\n");
                    break;
            }
            if (op != 0) print_list(&p);
    } while (op != 0);
                                                    /* détruit la liste : */
    destroy_list(&p);
    return 0;
}
```

Remarque **:**

La lecture des chaînes de caractères saisies par l'utilisateur est réalisée au moyen de la fonction fgets de la bibliothèque C standard.

La fonction fgets de la bibliothèque standard permet la lecture de chaînes de caractères saisies par l'utilisateur.

Cette fonction :

- lit une ligne complète sur le flux spécifié en troisième paramètre,
- stocke le résultat dans la chaîne fournie en premier paramètre,
- si le nombre maximal de caractères à lire est atteint, elle n'écrit pas le caractère terminal de la chaîne,
- elle stocke le caractère de saut de ligne en fin de ligne si ce nombre n'est pas atteint.

Il faut donc s'assurer que la ligne se termine par le caractère adéquat.

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32768 à 32767
unsigned short int	Entier court non signé	2	0 à 65535
int	Entier	4 (sur p 32 bits)	-2147483648 à 2 147 483 647
unsigned int	Entier non signé	4 (sur p 32 bits)	0 à 4294967295
long int	Entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	4	0 à 4 294 967 295
float	flottant (réel)	4	-3.4*10-38 à 3.4*1038
double	flottant double	8	-1.7*10-308 à 1.7*10308
long double f	lottant double long	10 -	3.4*10-4932 à 3.4*104932
bool	booléen		

11.Bibliographie

Livres:

C. Delannoy: Programmer en C++, Eyrolles, 2017.

S. Lippman, J. Lajoie, B. Moo: C++ Primer, Addison-Wesley Professional, 2012

K. Loundon: C++ Précis et concis, O'Reilly, 2003.

Sites internet::

http://www.cplusplus.com/

https://en.cppreference.com/

http://bruno-garcia.net/www/Cours/

http://casteyde.christian.free.fr/cpp/cours/online/book1.html

La consultation de ces références est vivement recommandée afin d'approfondir les notions abordées en classe.